A NUMA-Aware Version of an Adaptive Self-Scheduling Loop Scheduler

JOSHUA DENNIS BOOTH, Computer Science, The University of Alabama in Huntsville, Huntsville, United States
PHILLIP LANE, Computer Science, The University of Alabama in Huntsville, Huntsville, United States

Parallelizing code in a shared-memory environment is commonly done utilizing loop scheduling (LS) in a fork-join manner as in OpenMP. This manner of parallelization is popular due to its ease to code, but the choice of the LS method is important when the workload per iteration is highly variable. Currently, the shared-memory environment is evolving in high-performance computing as larger chiplet-based processors with high core counts and segmented L3 cache are introduced. These processors have a stronger non-uniform memory access (NUMA) effect than the previous generation of x86-64 processors. This work attempts to modify the adaptive self-scheduling loop scheduler known as *iCh* (*i*rregular *Ch*unk) for these NUMA environments while analyzing the impact of these systems on default OpenMP LS methods. In particular, *iCh* is as a default LS method for irregular applications (i.e., applications where the workload per iteration is highly variable) that guarantees "good" performance without tuning. The modified version, named *NiCh*, is demonstrated over multiple irregular applications to show the variation in performance. The work demonstrates that *NiCh* is able to better handle architectures with stronger NUMA effects, and particularly is better than *iCh* when the number of threads is greater than the number of cores. However, *NiCh* also comes with being less universally "good" than *iCh* and a set of parameters that are hardware dependent.

CCS Concepts: • **Software and its engineering → Runtime environments**; • **Computer systems organization → Multicore architectures;**

Additional Key Words and Phrases: Loop scheduling, irregular applications, performance evaluation, OpenMP

ACM Reference Format:
Joshua Dennis Booth and Phillip Lane. 2024. A NUMA-Aware Version of an Adaptive Self-Scheduling Loop Scheduler. *ACM Trans. Arch. Code Optim.* 21, 4, Article 75 (November 2024), 22 pages. https://doi.org/10.1145/3680549

1 Introduction

While many advanced parallel computing paradigms focus on complex tasking systems, the most common paradigm for application programmers is loop scheduling (LS) in a fork-join manner.

This work used Stampede2 at TACC through allocation CCR180052 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. Additionally, this work was supported by National Science Foundation grant #2135310.
Authors' Contact Information: Joshua Dennis Booth, Computer Science, The University of Alabama in Huntsville, Huntsville, Alabama, United States; e-mail: jdbst21@gmail.com; Phillip Lane, Computer Science, The University of Alabama in Huntsville, Huntsville, Alabama, United States; e-mail: phillip.lane@uah.edu.

ACM 1544-3973/2024/11-ART75
https://doi.org/10.1145/3680549

The commonality of this paradigm is due to the simplicity of coding, but the efficiency of this method can be less than optimal due to variations in the workload (i.e., the time for computation and memory accesses) between iterations. Because of the importance of LS, numerous coding languages and extensions offer a parallel-for (e.g., OpenMP) and research has been conducted on different methods (e.g., workload-aware self-scheduling) and parameters (e.g., chunk size) to achieve reasonable performance. In this article, we define chunk size as the amount of work (i.e., the number of iterations) a thread takes to execute from a queue before returning for more work. However, all of these methods require some information about the workload to select the best method parameters. This requires an understanding of the workload or tuning that a standard application developer might not have. A simple example of this would be a sparse matrix-vector multiplication (SpMV) parallelized over the rows (see Listing 1). In this example, a sparse matrix could have a variety of different non-zero patterns that are unknown before runtime and where the workload per iteration is the number of non-zeros in a row.

The scheduling method *iCh* (*i*rregular *Ch*unk) [4] was introduced as an adaptive self-scheduling method. The targeted workloads for this method are irregular problems where the workload per iteration varies in time due to both the number of computations and memory accesses. *iCh*'s goal is to not require workload information prior to runtime while providing reasonable performance (i.e., close to the optimal) for common multi-core systems with a shared L3 across a wide range of applications.

These common multi-core systems could have non-uniform memory accesses (NUMA) (i.e., the time to access local memory is faster than non-local memory). However, within a processor (i.e., a socket on the motherboard), the time for two different cores to access a line in a shared L3 was similar. Therefore, the only time that NUMA effects (i.e., the time difference in accessing local and non-local memory) were normally large enough to consider was between sockets.

Shifting architecture trends are resulting in "chiplets" and "segmented" L3 caches (e.g., AMD EPYC and Intel Sapphire Rapids) that have a strong NUMA effect on L3. In particular, the time for two different cores to access a line in L3 could be different depending on the core and the location of the line [24]. As such, the consideration of only NUMA effects at the socket level is no longer valid and therefore may impact the performance of *iCh*. Therefore, this article presents a NUMA-aware version of *iCh* (*NiCh*). As this architectural trend is just emerging, a uniform vocabulary has not yet been adopted as it has in the past with package, socket, and core. In this work, we will thus utilize AMD's terminology of chiplets, as they are the first major processor producer being highly utilized in high-performance computing with this trend. We note that this can be applied to the newer Intel systems or even IBM's Power series as well. This NUMA effect within a process between the chiplets now produces levels of NUMA effects. Section 2.2 provides more details on this trend as well as examples of NUMA access times. Even though we stick with AMD terminology and tests on AMD chiplet-based systems, our goal is that information learned from *NiCh* will be utilized across vendors as these chips become more widely available.

As this architectural trend could put *iCh* at a disadvantage, the following questions arise. Does *iCh* still provide a best solution for irregular workloads on newer architectures? Would a NUMA-aware version of *iCh* (*NiCh*) perform better than *iCh* on these new architectures? Would default OpenMP LS methods be a better solution now? This work attempts to shed light on these questions by providing the following:

— An introduction to *NiCh* and explanation of design choices
— An empirical evaluation of design choices of *NiCh*
— Comparison of *NiCh*, *iCh*, guided, dynamic, and static on the chiplet-based AMD EPYC 7713 (Milan)

```
1  function spmv(n, rowptr, colidx, val, x, y) {
2      #pragma omp parallel for schedule(runtime)
3      for(int row = 0; row < n; row) {
4          for(int c = rowptr[row]; c < rowptr[row + 1]; c++) {
5              y[row] += val[c] * x[colidx[c]];
6          }
7      }
8  }
```

Listing 1. SpMV where the matrix is provided in compressed row format. This demonstrates the imbalance of workload in a common kernel.

— Comparison on the more traditional Intel Xeon Platinum 8160 (Skylake)
— Comparison on Intel Xeon Phi 7250 (Knight's Landing)
— Analysis of sensitivity parameters of *NiCh*.

2 Background

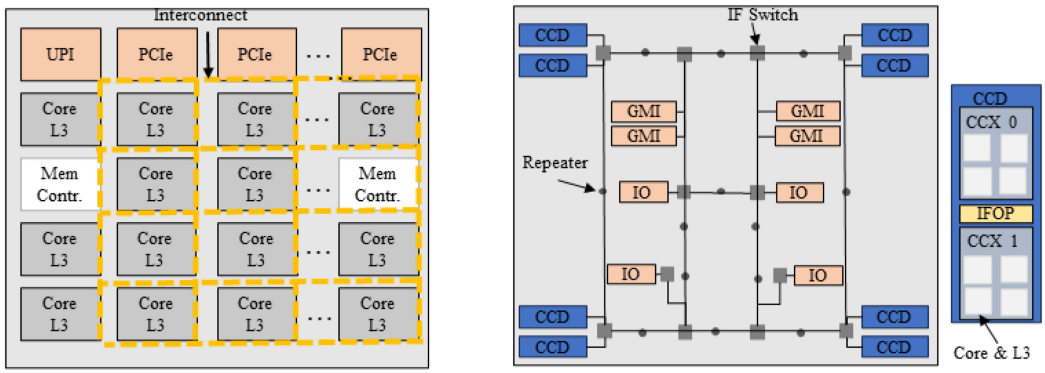
2.1 Loop Scheduling

The LS problem requires scheduling a set of n independent tasks (i.e., iterations) x_i, where $i \in \{1, \ldots, n\}$ onto p threads t_j, where $j \in \{1, \ldots, p\}$ in a way that minimizes the total parallel execution time. The workload of each task x_i can be different and unknown at runtime. Theoretically, this type of problem is NP-hard [6]. However, this problem is normally even harder than what the theoretical problem considers on modern systems, as the time to complete x_i can vary between executions as frequency scaling can be applied to cores and NUMA effects can impact memory access time. Consider the SpMV example in C/OpenMP (Listing 1). Although the algorithm looks simple on the surface, its memory characteristics have the potential to be highly irregular. This is due to the irregular access of the `x` array. If consecutive values in `colidx` are vastly different, this can result in sporadic, pseudo-random accesses in `x`, which results in unpredictable performance as one access may be in an L1 cache line, whereas the next requires a traversal to main memory [14]. Making matters even worse, the memory access patterns of `x` are entirely dependent on the input, meaning that they can change drastically between runs.

Broadly speaking, three different approaches can be taken to solve the problem.[1] The first approach is to remove the unknown aspect of workload. LS methods in this first approach try to build some understanding about the distribution of the workload either through analysis of the workload before runtime (e.g., via mini runs or code analysis at compile time) or by building up a history of the workload when multiple iterations are used [16]. This approach has been shown to be effective in some applications but normally requires some overhead for the analysis or history construction, and this approach may fail if the workload varies a lot in a manner the analysis cannot identify. While no standard OpenMP method fits this approach, numerous packages have studied it (e.g., COWS [19] and BinLPT [22]).

The second approach is for the method to try to adapt at runtime to the workload. The most common pattern for these methods is to either have a shared queue where threads can take work as needed or have threads steal work from each other as needed. However, overheads exist in utilizing a shared data structure in a thread-safe manner and identifying where to steal. This approach is somewhat similar to the default *dynamic* approach by OpenMP, which utilizes a single shared queue, and many work-stealing packages (including *iCh*), which utilize multiple double-end queues (i.e., deques) to reduce thread contention.

[1]There exist more than three classifications in LS taxonomy (see [4]), but only three general approaches need to be understood for this work.



(a) ICL (b) Rome

Fig. 1. Two example architectures: ICL and AMD EPYC Rome. The ICL utilizes a common 2D mesh connecting cores with L3 cache to memory controllers (Mem Contr.) and PCIe nodes on the same chip. Rome utilizes chiplet and uses a complex network of IF switches and repeaters to connect cores contained in CCDs with global memory interface (FMI) and IO.

The last approach is to construct a very simple method with a parameter that is tuneable to the workload. The concept is that the method is so simple that any imbalance in the workload is negligible compared to the overhead of the other approaches. Although this assumption is fine for some workloads, it does not apply to all. Some of these methods utilize a fixed chunk size (e.g., *static* that is a default in OpenMP). Others provide a variable chunk size. Examples of these include *guided* found in OpenMP and other common methods such as factoring [1] and trapezoid [23], which have appeared often as an alternative to *guided*. Unfortunately, the need to keep the method simple normally requires these non-fixed chunk size methods to use a simple method of always decreasing the chunk size by some fixed amount. Work by Kasielke et al. [15] shows that *guided* is still the best of these simple non-fixed chunk size methods. These simple methods tend to do very well as we report in Section 6 with *guided* and COWS [19] reports with *cyclic*. We note that *cyclic* is a special case of OpenMP *static* where the parameter is set to 1 (therefore, we also test with this version along with the default). In the default *static*, the n iterations are chunked up to the p threads in loop order. With *cyclic*, the n iterations are evenly distributed to the p threads in a round-robin manner. This distribution helps to even out some of the imbalance in the workload among the threads. However, a quick counter-example exists. Consider a for-loop where even numbered iterations do x_{even} amount of work and odd numbered iterations do x_{odd} amount of work. When the program has an even number of total iterations and the workload is divided among an even number of threads, there will exist an imbalance among the even and odd threads. This is but one example where simple methods can fail, and the goal of *iCh* and *NiCh* is to be good in most cases.

2.2 Architecture Trends

Complex chiplet-based processors are becoming the norm with examples including AMD EPYC and Intel Sapphire Rapids. These chiplet-based processors are introducing NUMA effects that have been unseen before by traditional multi-core processors. Consider the following example comparing the performance of AMD EPYC Rome and Intel Cascade Lake SP (*ICL*) server processors. Figure 1 provides the layout of both. The older ICL processor has the core and L3 segments laid out in a 2D mesh on the same chip. This makes communication and memory accesses between the cores quick and efficient, even though some NUMA effects may exist. In comparison, the layout of Rome

is quite different but is still composed of multiple chiplets. The layout has sets of Core Complex Dies (CCDs) that are connected via AMD's Infinity Fabric (IF). Each CCD contains one or more Core Complex (CCX). Each CCX contains multiple cores (normally four) along with each core's associated L1, L2, and L3. Routing between cores in different areas requires movement through repeaters and IF switches. In a way, the communication on the ICL is like homogeneous packet-switching with a known route (i.e., all accesses are routed down and over). However, the communication in Rome is closer to a heterogeneous packet-switching with an unknown route (i.e., routing is similar to routing through switches in a local area network). Velten et al. [24] measure the latency reads for different cores for both of these architectures. For reading the L3 cache, the ICL latency from local to distant neighbor is less than 20 cycles difference. However, the difference in Rome from the local to the distant core can be ~200 cycles. As such, this is a much stronger NUMA effect.[2] Thus, this effect makes considering what is the best LS method very important. Moreover, as AMD tends to have the strongest effect at the moment, *NiCh* is initially designed around this architecture.

3 *iCh* Algorithm Overview

This section provides a summary overview of the *iCh* algorithm with the goal of providing the reader enough background in *iCh* to understand the modifications made in *iCh* to produce *NiCh*. We invite readers to reference the original work for *iCh* for more insight into design decisions and a running illustrated example [4]. In keeping with LS literature convention, both *iCh* and *NiCh* are explained in terms of threads. However, we assume that a particular thread is pinned to a core, as this allows us to facilitate a conversation of NUMA effects.

The *iCh* algorithm is an extension of traditional work-stealing. The traditional work-stealing algorithm evenly allocates the n iterations among the p threads into private deque (i.e., one deque per thread). When a particular deque runs out of work, the thread randomly selects a deque to steal half of the remaining iterations. Although relatively simple in design, the algorithm has the ability to be 2-*approximate* [6]. On a relatively small set of shared-memory threads, the traditional work-stealing algorithm performs well on irregular applications [9, 12]. However, a couple of implementation details tend to get in the way of performance as the number of threads increases. The first of these issues is that randomly selecting the thread to be stolen from can be expensive. The reason for the expense is that a thread tries to steal from a thread that has no iterations left or few remaining iterations. This means that the thief (i.e., the thread performing the stealing) will have to try again, spin wait, or steal again soon if few iterations remain. Some works (e.g., [19]) have tried to build some structure to allow prediction or analysis on which thread should be stolen from. However, most works have not shown much success because of overhead in pulling from a shared data structure for the analysis or some pre-runtime analysis that is required. The second issue is that the chunk size (i.e., the active set of iterations the thread is currently working on) from its own queue is fixed. A tradeoff space exists in regard to chunk size. As the chunk size increases, the less the thread has to come back to its private deque to get more iterations of work. However, a larger chunk size leaves fewer iterations in the queue for a thief to steal. There have been many works related to how to make the stealing process less likely to interfere in regard to locks on this shared private queue to mitigate the issues with chunk size [12].

iCh has taken a different approach to the chunk size issue facing traditional work-stealing algorithms. The approach used by *iCh* is to have a local adaptive chuck size that will be updated based on the flow of iterations. In particular, the chunk size is made smaller for threads that are processing faster, as the overhead of having a smaller chunk size would be less impactful. To

[2]Current non-peer reviewers have stated that the newer generation of chiplet-based processes has slightly reduced this NUMA effect (https://www.anandtech.com/show/16529/amd-epyc-milan-review/4).

describe this in more detail, we break *iCh* into three phases: initialization, local adaption, and remote work-stealing.

3.1 Initialization

iCh uses local deques for each thread denoted as q_i, where $i \in \{0, 1, \ldots, p-1\}$ is the thread ID of the p threads. The local data structure is allocated using a first-touch allocation policy and is memory aligned. The structure also contains a local counter (k_i) and a variable used to calculate chunk size (d_i). The variables k_i and d_p are initialized to 0 and p, respectively. With these variables, the resulting initial chunk size is n/p^2 (i.e., $chuck\ size = |q_i|/d_i$). This size is picked to allow for multiple steals from the other $p-1$ threads. We note that the rationalization of utilizing one deque per thread is threefold. First, a thread would have no contention with other threads unless being stolen from. Second, more information from the data structure can be kept in higher levels of cache without the need to write back to L3 for sharing with other threads. Third, work for a chunk is always taken from the tail end of the deque, whereas iterations are only taken from the head end during the remote stealing phase. This allows for reduced locking and better memory consistency.

3.2 Local Adaption

In this phase, the *iCh* method attempts to improve upon traditional work-stealing by locally adapting chunk size to better fit the running distribution of iterations completed across threads. In local adaption, a thread first classifies its computational load relative to other threads as

$$\text{low: } k_i < \sum_{j=0}^{p-1} k_j/p - \delta, \tag{1}$$

$$\text{normal: } \sum_{j=0}^{p-1} k_j/p - \delta \leq k_i \leq \sum_{j=0}^{p-1} k_j/p + \delta, \tag{2}$$

$$\text{high: } k_i > \sum_{j=0}^{p-1} k_j/p + \delta. \tag{3}$$

This classification is similar to the standard range around the (μ)—that is,

$$\mu - \delta \leq \mu \leq \mu + \delta, \tag{4}$$

where δ is some multiple of the standard deviation (σ). In particular, $\mu = \sum_{j=0}^{p-1} k_j/p$, which is the mean number of iterations complete per thread or otherwise called *mean iteration throughput*.

As with any distribution range around the mean, the δ is important. In our case, this δ attempts to capture the essence of the variance. This variance in turn attempts to capture how dispersed the number of floating-point operations and memory requests are from the mean. However, calculating the exact running mean and variance can be difficult. In particular, the running approximation exists [25]:

$$\mu_{i+1} = \mu_i + (k_i - \mu_i)/p, \tag{5}$$

$$\sigma^2_{i+1} = \sigma^2_i + (k_i - \mu_i)(k_i - \mu_{i+1}), \tag{6}$$

where i represents the timestep. As this approximation is too expensive, *iCh* estimates this value with a fractional multiplier (ϵ) of the running mean—that is,

$$\delta = \epsilon \sum_{j=0}^{p-1} k_j/p. \tag{7}$$

As a result, the variation or standard deviation would be a multiple of the running mean, and this multiplier would provide a way to either tighten (i.e., make the interval smaller) or loosen (i.e., make the interval larger). The δ will also grow with iterations completed. The relationship between δ and the number of iterations completed will result in *iCh* being more likely to adapt the chunk size in the beginning and less likely at the end. The logic behind this decrease chance of adapting the chunk size would fit the idea that the chunk size would not need to be updated as often toward the end of execution when most of the information (i.e., workload) has already been assessed. The only other parameter that needs to be considered is the user-provided parameter ϵ. This scaling factor (i.e., $0 < \epsilon < 1$) has been shown in the past to not be very sensitive to the input [4].

After classification into low, normal, or high, the chunk size is adapted. The parameter d_i is used to calculate the chunk size (i.e., $chunk\ size = |q_i|/d_i$), as follows. For a low classification, $d_i = d_i/2$, and the chunk size increases (i.e., $chunk\ size = |q_i|/(d_i/2) = 2|q_i|/d_i$). For high classification, $d_i = 2 \times d_i$, and the chunk size decreases. The rationale for the updating d_i is as follows. In the low classification, the thread is completing fewer iterations than the mean. One reason a thread could be completing fewer iterations is that the chunk size is too small. A second reason could be that the thread takes more time per iteration than other threads, possibly due to dynamic voltage and frequency scaling (DVFS) variations. Based on these reasons, *iCh* should assign a large chunk size to make stealing less likely and engage the thread in more work. In contrast, a high-classified thread has completed more iteration than the mean. This thread would have more free cycles to update chunk size or deal with stealing requests. We note that the direction in which chunk size is updated is in opposition to the logic that one might have if optimizing to balance the average amount of work assigned to each thread in a chunk.

3.3 Remote Work-Stealing

When the local deque runs out of work, the thread must steal from some other thread. The *iCh* algorithm uses the same random stealing algorithm as other work-stealing methods to identify the victim and steal half the remaining iterations [9, 12], namely the THE protocol. This allows for relatively few locks, and the stolen iterations are taken from the head of the thief's deque to reduce interference with its current chunk and improve data locality between chunks. However, unlike other work-stealing methods, *iCh* must update the parameters related to adaptive chunk size. The *iCh* assumes that both sets of parameters (d_{thief}, k_{thief}) and (d_{victim}, k_{victim}) contain some valid information about either the workload of the iterations in the local deque or the rate in which the thread can process (e.g., the thread is scheduled to a low-voltage core that would reduce the speed in which computations can be executed). As such, *iCh* averages the information from both the thief and the victim to update the d and k of the thief as follows:

$$d_{thief} = (d_{thief} + d_{victim})/2, \tag{8}$$

$$k_{thief} = (k_{thief} + k_{victim})/2. \tag{9}$$

The d and k for the victim remain the same. The next chunk size will be calculated using the updated d_{thief} and k_{thief} in the same manner that is outlined in the local adaption phase. We note that this is not the most elegant method for updating but can be done cost-effectively. Moreover, the original work of *iCh* observed that this was better than not updating d and k.

4 *NiCh* Algorithm Overview

This section outlines the design and architecture of the *NiCh* algorithm. The *NiCh* algorithm is directly based on *iCh* except for several modifications that aim to reduce the NUMA effect overhead. In particular, these modifications are made with the AMD EPYC in mind, as this architecture has

large NUMA effects and is common in high-performance computing (see Section 2 for information on AMD Rome and Section 5 for information on AMD Milan). However, these changes could also be made for any future chiplet-based architectures that are coming out from other vendors like Intel. Within each phase, we provide specific guidance on why these changes were made for the target architectures to aid changes for new chiplet-based architectures as they become available.

The changes to the *iCh* can be summarized as follows:

(1) Number of threads per deque (initialization). *iCh* utilizes one deque per thread for reasons stated before. The chiplet structure limits the number of threads that would be accessing the shared L3 segment and could provide a better method for sharing information. As such, threads within a chiplet structure could share a deque. However, we note that there is some additional overhead for this, as more locking needs to take place to ensure correctness that the THE protocol does not address for private deques.
(2) How often chunk size is updated (local adaption). *iCh* updates the chunk size of a deque each time a thread returns for more work. *NiCh* considers updating the chunk size less often due to the increase in time of collecting information from all threads to update the chunk size and due to more threads utilizing the locally shared deque.
(3) Stealing in a NUMA-aware manner (remote work-stealing). *iCh* randomly steals from any other thread. *NiCh* utilizes a two-level stealing approach that tries to first steal from its own chiplet region (i.e., CCD).

Details of these three changes are provided in the three phases used by *NiCh*.

4.1 Initialization

As in *iCh*, the initialization sets up deque and key parameters. The difference in *NiCh* is that private deques for each thread are not utilized. A number of shared deques are set up in a NUMA-aware manner (i.e., cores that are close share a deque). In addition to the shared deque, these threads will also share the parameters of d_q and k_q, where q is the particular deque, and thus they will share the same chunk size. The total number of iterations are evenly distributed among the deque. A shared deque has the disadvantage of possible thread contention, but shared deques in a single NUMA region with relatively few threads sharing would have more advantages than disadvantages. In particular, these shared deques are designed with the following advantages in mind: shared deques allow local threads to avoid stealing, and they can communicate some of their information into the shared d_q and k_q parameters. While stealing between local threads should be fast since they are in the same NUMA region, other overheads exist with stealing, such as possibly selecting a deque outside of the local NUMA region and obtaining a global approximation of the mean. The number of threads associated with a local deque (p_q) is one more sensitive parameter that is added to the algorithm (in addition to ϵ). However, p_q should be on the order with the number of threads in a NUMA region.

AMD Milan Target. For AMD Milan, the target is to allow a shared deque between every set of four cores. This results in two shared deques per CCX and CCD, and four shared deques per quadrant. Stealing between the two shared deques on a CCD is fast because all eight cores in a CCX have near-uniform access to a shared pool of L3 cache. Stealing between the queues on the same quadrant across CCDs would be significantly slower because a request must be made from one CCD across the data fabric, through the IO die, and to the other CCD on that quadrant. Then, the requested data must make the trip back to the requesting CCD. Requesting anywhere else would be even slower, as there is a significant physical distance that must be traveled through the IO die to get to other quadrants.

```
1  function next:
2      label retry:
3       if try_local_work()->start,end:
4         return start,end
5      if stealing: //check local thread stealing
6         goto retry
7      else:
8         stealing = true
9         do_steal()
10        stealing = false
11        goto retry
12 function try_local_work:
13    chunksize = get_chunk_size
14    //remove from the tail end
15    lock(local)
16    start = local.start
17    local.start = local.start + chunksize
18    end = local.start
19    local.update_count++
20    local.icount += chunksize
21    unlock(local)
22    if local.update_count > KUpdate:
23       local_adaption()
24 function local_adaption:
25     sum = sum_all_icount()
26     sum /= local.nlocals
27     alpha = sum x local.epsilon
28     if local.icount < sum-alpha:
29          local.ki << 2
30     elif local.icount > sum+alpha:
31          local.ki >> 2
32     local.update_count = 0
```

Listing 2. Pseudocode for Nich local adaption phase. The next function tries to retrieve the next chunk of work by returning the start and end of the chunk. The try_local_work function calculates the current chunksize and tries to get the next chunk of work. The local_adaption function updates the chunksize. Key user parameters are bolded (e.g., KUpdate and epsilon).

4.2 Local Adaption

The algorithm for the iteration and local adaption is provided in Listing 2. The same general selection of cases (i.e., low, normal, and high) and change to chunk size are implemented in *NiCh* as in *iCh* (lines 24–32). Originally in *iCh*, the chunk size is updated after every chunk of iterations that is completed (lines 22 and 23, i.e., `local.update_count = 0` for *iCh*). However, the question of how often the chunk size should be updated arises due to two factors. The first of these factors is the overhead of producing the approximation of a mean. The shared queues reduce the number of k_q that need to be globally summed, but in most cases, the calculation will be very expensive due to NUMA effects. The second of these factors is that the chunk size could "ping-pong" between completed chunks as it becomes biased to the last thread that was completed. For example, consider the case of a shared queue with two threads. One thread could be very fast due to workload or frequency, and the other thread could be very slow. As the fast thread gets done, it will keep updating the shared chunk size into smaller pieces. However, when the slow thread is done, it will most likely have to take a small chunk even though it is not the best choice. This introduces one more sensitivity parameter, $KUpdate$—a parameter indicating the number of times the local queue

chunk size should be used before updating. Additionally, we note that locks (lines 15 and 21) are required in *NiCh* as we share the deque but not required in *iCh*.

AMD Milan Target. Due to the target thread per queue count of four, the parameter *KUpdate* is also set to four. In essence, if all four threads in a queue complete their work, the last one to complete will update the d_i.

```
1  function do_steal:
2      victim = pick_victim() //in a NUMA-aware manner
3      size = (victim.end - victim.start) << 2
4      lock(victim)
5      victim_list = steal_from_head(victim,size)
6      unlock(victim)
7      lock(local)
8      add_to_head(local, victim_list)
9      local.icount = (local.icount + victim.icount) << 2
10     local.ki = (local.ki + victim.ki) << 2
11     unlock(local)
```

Listing 3. Pseudocode for the Nich remote work-stealing phase. The pick_victim function will randomly pick victims in a hierarchy-based manner where the hierarchy is defined by the NUMA regions. In particular, it tries to steal from close NUMA regions first before moving to other NUMA regions. The steal_from_head function removes the iterations from the victim's deque from the head.

4.3 Remote Work-Stealing

The algorithm utilized for remote work-stealing is provided in Listing 3. In *iCh*, a victim is selected at random among all threads (line 2). The method is modified to use hierarchical stealing where the hierarchy is constructed in a NUMA-aware manner (i.e., levels are constructed based on NUMA distance) for *NiCh*. The concept of utilizing hierarchical stealing for work-stealing is not new [9]. However, the use of it for works stealing LS in the past has not had much of an impact on a single processor due to the flat cache access structure [9]. The values of d and k are updated in the same manner as with *iCh* with the thief averaging the value from the victim (lines 9 and 10). Additionally, we note that in comparing the algorithm for *iCh* and *NiCh*, more locks are needed to be in place (e.g., line 7) because of the shared deques that are used.

AMD Milan Target. For targeting AMD Milan, the following local is utilized for hierarchical stealing: because the two queues per CCX share quick access to the same L3 when a queue first attempts a steal, it will try to steal from the other queue on the same CCD for a low-latency steal. If this fails, it will randomly attempt to steal from queues across the processor without discrimination until it is successful. Upon a successful steal, the state is reset and a further steal would attempt to steal from the other queue on the same CCD, repeating this process. Therefore, only a two-level hierarchy is utilized for picking the victim (line 2).

5 Testing Setup

In this section, we lay out our testing setup.

5.1 Computer Architecture

The original work of *iCh* is evaluated on two Intel Xeon E5-2695 v3 (Haswell) processors each with 16 cores and 128 GB of DDR4-2133 [4]. Although this system demonstrates NUMA effects, they are relatively mild compared to some of the architectures of interest in high-performance computing. As such, we test on three systems listed in the following with stronger NUMA effects.

AMD Milan. The first system contains the third-generation AMD EPYC 7713 (Milan), which we denote as *AMD* in Section 6. Figure 1(b) provides the layout of the second-generation AMD Rome as an example. The system is located at the Alabama Supercomputing Authority in the Dense Memory Cluster.[3] These chips have a chiplet construction in the following manner. The base unit is a CCX, and each CCX contains eight cores and shares an L3 cache. A CCD contains one CCX.[4] A bi-directional ring bus connects the CCDs, and the bus allows up to 32 bytes of data per cycle when receiving or 16 bytes of data per cycle when sending.[5] The L3 cache is not shared between CCXs and is only shared within a CCX.[6] As for AMD Rome, if a core needed to access a cache line in the *shared* state that resided in a different CCX, the core would instead request the data from main memory [24]. It is safe to assume that this behavior continues for Milan. This indicates that inter-CCX communication is primarily for cache coherence, but since the ring bus also carries data to and from main memory, there is the potential for contention over the bus. As this is the newest and most extreme in regard to NUMA effects, it is our primary architecture when tuning *NiCh* originally. For our particular processor, the base clock speed is 2.0 GHz with a max clock speed of 3.675 GHz. This large difference in clock speed is one of the issues that a workload scheduler must deal with, as different threads may be running at different rates. Although the system supports up to 128 threads, we do not test above 64, as results became incomprehensible for all runtimes. Part of this may be due to the fact that inter-CCX communication on socket 1 has twice the latency of inter-CCX communication on socket 0 [7]. Therefore, we restrict testing to socket 0 only. The total L3 size is 256 MB. All codes were compiled with GCC 8.2.0.

Regarding the extreme NUMA effects between CCDs, we note that besides the large physical distance that must be traveled, one of the primary reasons that inter-CCD communication is slow is that AMD uses a central hub for all communication. In essence, the CCDs can only communicate via routing through the IO die [24]. The newer Intel Sapphire Rapids systems opt instead to use an all-to-all bus for inter-die communication [20]. Even though being all-to-all can be advantageous for inter-die communication, it fundamentally limits how many tiles can be glued together and as such contributes to the peak core count of 56 cores. This is in contrast to the centralized approach of Milan and other AMD architectures, which leads to easier scaling of processor dies at the cost of slower and higher latency inter-die communication.

Intel KNL.. The Intel Xeon Phi 7250 (*KNL*) has a unique architecture that was popular for a short time in high-performance computing. The KNL has 68 cores with a base frequency of 1.4 GHz. Each core can support up to four hardware threads (although all four threads are rarely used in practice). It has a private L1 and a 1-MB L2 per two-core tile. In place of a standard L3, the KNL has a 16-GB MCDRAM that can be set up in several different modes. The most common mode for the MCDRAM is a direct-mapped L3, which is the mode used in this work. Despite waning interests in the architecture and removal of it from Intel's line, the unique architecture is ideal for testing such a runtime scheduler. The reason for this is because of the large number of cores that communicate with the MCDRAM in a 2D mesh interconnect, which results in a NUMA effect based on location in the 2D mesh. In addition, the MCDRAM is separated into quadrants, meaning that certain groups of cores can access particular memory regions in MCDRAM faster than other cores based on which MCDRAM quadrant a core lies within. All codes on this system were compiled with GCC 7.3.0. This system was utilized on Stampede2 at TACC [3].

[3] https://www.asc.edu/service-area/high-performance-computing

[4] The fact that a CCD only has one CCX differs from the older Rome architecture that had two CCXs per CCD.

[5] Information obtained via correspondence with AMD.

[6] This fact about the shared L3 is normally glossed over in the marketing and technological manuals for this processor, although the next generation is marketing their stacked cache for L3.

Intel Skylake. The last system considered is an Intel Xeon Platinum 8160 (*SKX*). This system does not have as interesting a design as the AMD and KNL, but it is a newer Intel chip than used by the original *iCh* work with more cores and different architecture. This system has 48 cores in two sockets (24 cores/socket), and each core can support up to two hardware threads. The clock rate is 2.1 GHz but can be adjusted to 1.4 to 3.7 GHz depending on the instruction set and number of active cores. Therefore, testing on this system while utilizing more than one hardware thread would provide various computing power that would be ideal for testing various scheduling methods. Each core has a private L1, a private 1-MB L2, and a shared 33-MB L3 per socket. All codes on this system were compiled with GCC 7.3.0. This system was utilized on Stampede2 at TACC [3].

5.2 Tested Runtimes

Both *iCh* and *NiCh* are implemented inside of GNU libgomp under the GPL v3 License. The choice of compilers outlined in the previous section is done to align the closest with the original *iCh* work that was available on the system through the module system. OpenMP threads are bound to cores with `OMP_PROC_BIND=true` and `OMP_PLACES=cores`. The OpenMP schedules of *static*, *dynamic*, and *guided* are tested against *iCh* and *NiCh*. Chuck size parameters for *dynamic* and *guided* include 1, 2, and 3, and only the best result is reported in Section 6. We note that the original work of *iCh* compared against a generic work-stealing algorithm, OpenMP's *taskloop*, and BinLPT, in addition to *dynamic* and *guided*. We chose to leave off the first three, as only generic stealing did well overall, and *iCh* normally outperformed generic work-stealing. Moreover, we added *static* (i.e., default and cyclic scheduling with parameter 1), as this is more likely to be used by an application user [19] and has surprising results for our test systems.

5.3 Applications

We consider seven applications for testing *iCh*, *NiCh*, and the other OpenMP runtimes. Four of these applications are used in the original work for *iCh* [4], although the sizes and parameters need to be adjusted to compensate for our current larger test systems. The applications are as follows, and we provide a table of estimated work per iteration for each application in Table 1. We note that since the type of work done for each application varies (e.g., memory access, integer operations, floating-point operations), comparison of the exact numbers between applications is difficult, but the numbers do provide some insight into the application's irregularity. More details on how these are computed per application are given in the following.

Synthetic. The first application is a synthetic application that was first presented in the work for the OpenMP runtime BinLPT [22]. This application is also presented in the original work of *iCh* [4], where *iCh* is tested against BinLPT. We do not provide timings for BinLPT, as *iCh* has already been proven to outperform BinLPT [4], but the synthetic application is well fitted for testing workloads with different distributions. We consider three different workloads. These workloads are Uniform (`Syn Uni`), exponential increase (`Syn Exp Inc`), and exponential decrease (`Syn Exp Dec`). Workload calculations provided in Table 1 count the number of integer operations that are done.

miniTri. This application (`Tri`) is part of the Exascale Computing Project proxy application in Mantevo [10, 21]. The goal of the application is to use triangle enumeration with a calculation of specific vertex and edge properties. As such, the application is a proxy app for standard graph-based data analytics such as Graph 500. The input of the proxy app is `hugetrace-00000`, which is a DIMACS10 [17] undirected graph with 4,588,484 vertices and 13,758,266 edges. The workload calculations provide in Table 1 are based on the number of non-zeros in the triangular enumeration matrix form in the first step of the algorithm.

Table 1. Estimate Work Per Iteration of Applications

Application	Mean	Variance	Range
Syn Uni	2E+0	0E+0	0E+0
Syn Exp Inc	1E+5	9.9E+9	1E+6
Syn Exp Dec	1E+5	9.9E+9	1E+6
Tri	6.3E+0	5.9E+0	8.4E+1
BFS Uni	4.8E+1	8.2E+1	1.1E+2
BFS Pow	8.3E+1	2.2E+4	1.2E+2
Kmeans	2.6E+2	1.1E+6	1.1E+6
LavaMD	8.8E+8	7.8E+17	1.9E+5
Path	6E+0	2E+-8	1E+0
LUD	5.4E+4	2.4E+7	1.6E+4

Breadth-First Search. The breadth-first search (BFS) in the Rondinia [5] benchmark suite is used. The Rodinia benchmark suite has several benchmarks for large heterogeneous systems. Two different distributions were utilized to generate graphs for the BFS utilizing a modified version of the graph generator that comes with the benchmark. The first distribution utilizes nodes that have a uniform distribution of the number of neighbors (BFS Uni), and this is the standard distribution generated for this benchmark. The second distribution utilizes the power-law distribution to generate a scale-free network (BFS Pow), and this distribution was added to the modified graph generator. Scale-free networks are those where the fraction of $P(k)$ nodes in the network having k connections is $P(k) \sim k^{-\gamma}$, where $\gamma = 2.3$ in our tests. These networks are common in areas such as social networks, computer networks, and protein interactions. Moreover, BFS tends to be a fundamental kernel in many graph analytic applications. For both workloads, approximately 16 million vertices are generated. We note that this can result in significantly more work because of the number of edges for the graph generated with the power distribution. The workload calculations in Table 1 are based on the number of edges that are generated for each node.

K-Means. The Kmeans (Kmeans) benchmark from Rodinia [5] is used because it is a common machine learning algorithm. The original *iCh* work utilizes the KDD Cups dataset related to network packets. However, a larger dataset is needed for this scaling study. For this study, a set of 1 million data points with 128 clusters is generated from the data generator provided by the benchmark. This benchmark is highly irregular in that the workload distribution in the innermost loop changes per outermost loop iteration. The workload calculations in Table 1 are based on the number of floating-point distances calculated, compared, or summed.

LavaMD.. LavaMD (LavaMD) is a computational fluid dynamics (CFD) code from the Rodinia [5] benchmark suite. This code utilizes an N-Body simulation to simulate the interactions of solidification of molten tantalum and quenched uranium. We use an input size of $16 \times 16 \times 16$ to construct the domain. Force calculations are done for particulars within the box at each step. Calculations done between boxes are based on a cut-off ratio of about the size of a box. Therefore, interaction calculations are only done with neighboring boxes. The workload calculations in Table 1 are based on the number of elements computed. The high variance comes from the difference in the number of elements computed in an interior block with many neighbors and an edge or corner block with few neighbors.

Pathfinder. Pathfinder (Path) is a graph transversal that utilizes a dynamic programming approach to find a path on a 2D grid with the smallest total weights. This benchmark is again from

the Rodinia benchmark suite. The `Path` benchmark takes in the parameters of the width for the 2D grid and the number of steps for dynamic programming. We use a grid width of 100 million and a step of 10, which tends to be the normal parameters for larger systems. This program is of particular interest because it is not embarrassingly parallel out of the box (unlike `Kmeans`) due to the dynamic programming elements. However, the work tends to be perfectly balanced as you walk the grid, as seen in the work calculation of Table 1 that considers the work done per grid element.

LUD.. Dense LU factorization (`LUD`) is highly optimized, as it is a key dense linear algebra kernel. Again, this benchmark is taken from the Rodinia benchmark suite. The normal algorithm for this tiles the matrix and computes the dense factorization of the diagonal block and sends updates to backsolve the off-diagonal blocks. While the work per block tends to be uniform in the dense case, the number of blocks updated changes for every outer iteration. This benchmark is chosen because even though it may be grouped into the category of regular kernels, the imbalance can still be ideal to show off the flexibility of *iCh* and *NiCh*. To demonstrate the irregularity of work, the work calculation in Table 1 is the number of blocks available to schedule instead of the work per block.

6 Results

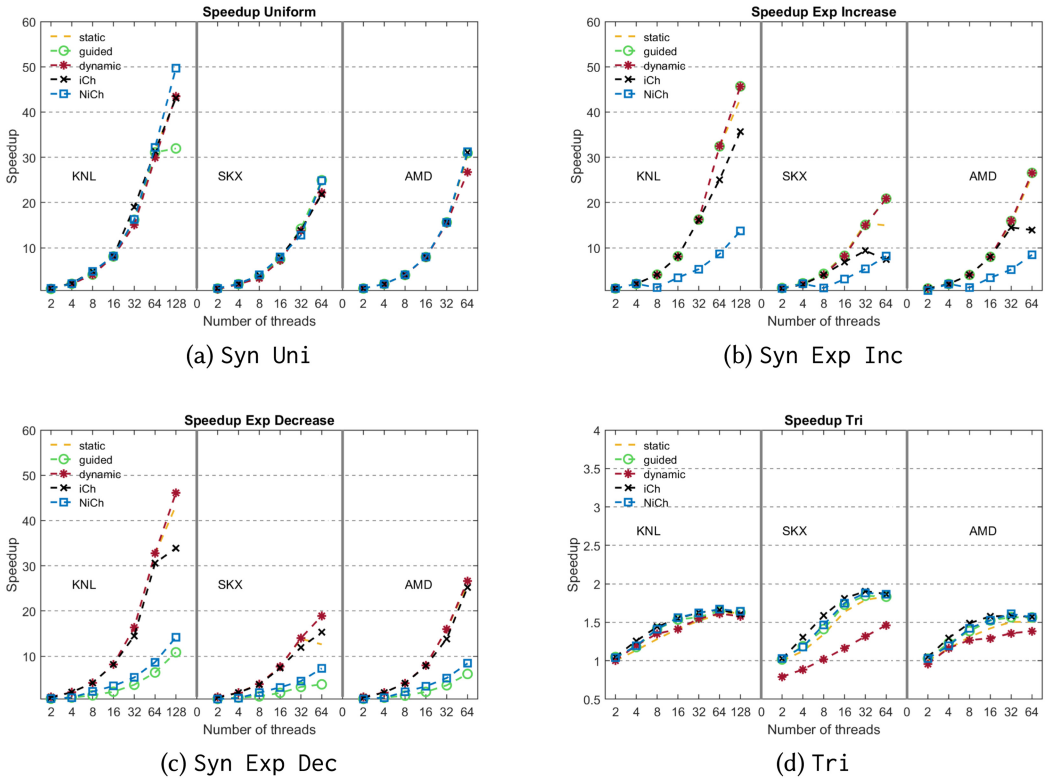
In this section, we present the empirical testing of *iCh* and *NiCh* as reported as speedup compared to the other standard OpenMP LS methods for our test applications. Here, the speedup is defined as

$$speedup(app, sys, sched, p) = \frac{time(app, sys, static, 2)}{time(app, sys, sched, p)}, \tag{10}$$

where $time(app, sys, sched, p)$ is the time for the application (app) to run on the system (sys) using the LS method ($sched$) with p threads. The $time$ that is reported is the best time for the $sched$ over all parameters tried. We use speedup over two threads, as some of the schedulers performed quite oddly under one thread, and thus we consider the results with one thread to be unusable. We define the base of comparison to be the best *static* method, as *static* is the default method in OpenMP.

Figure 2 provides the speedup figures for the `Synthetic` and `Tri` applications. Each plot provides the speedup for the application on the three different systems. From left to right, the systems are KNL, SKX, and AMD. The three `Synthetic` applications give a good sense of how well the scheduling methods work on well-known discrete distributions of workload. We note that the goal of both *iCh* and *NiCh* is really for applications that have irregular distributions of workload. The uniform distribution (`Syn Uni`) provides a good indicator of any scheduling overheads that might exist, and we would expect *static* scheduling to be ideal for this case. For this distribution, we observe that all scheduling methods do about equally well. The only points of diversion are at the ends when the number of threads is either greater than the number of cores (i.e., KNL and SKX) or full utilization of the cores (i.e., AMD). In these endpoints, *NiCh* is able to slightly outperform the others. In contrast, the `Syn Exp Inc` application demonstrates a case where *NiCh* does significantly worse than all LS methods. However, the less intelligent method of *iCh* does better than *NiCh* when the number of threads is equal to the number of cores for KNL and SKX. This trend continues into the `Syn Exp Dec` application with the addition that *guided* also does slightly worse than *NiCh*. In the previous *iCh* work [4], *iCh* has about the same trend for the `Synthetic` application on the Intel Haswell system.

In contrast to the `Synthetic` application that has a very well defined distribution, the `Tri` application in Figure 2(d) has a very irregular distribution of workload. The LS for this application is known to be difficult to scale. We observe that both *iCh* and *NiCh* do equally well on this



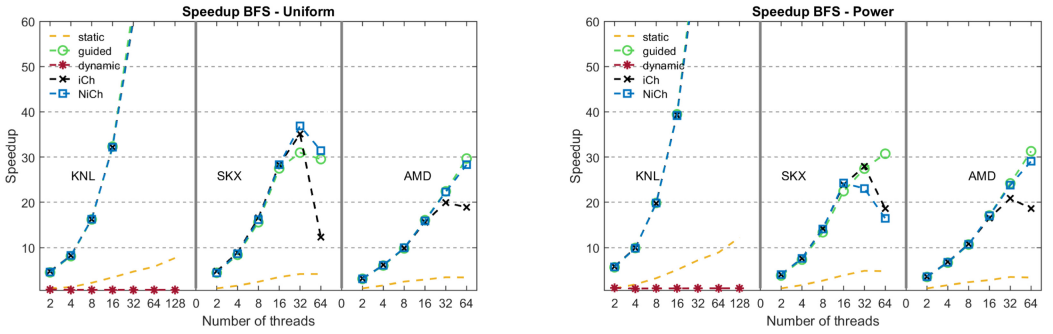
(a) Syn Uni (b) Syn Exp Inc

(c) Syn Exp Dec (d) Tri

Fig. 2. Speedup for the Synthetic and Tri applications. Most LS methods do about the same on the Synthetic application; however, *NiCh* has less than desirable performance that matches *guided* for more complex distributions. The Tri application demonstrates where *dynamic* does poorly.

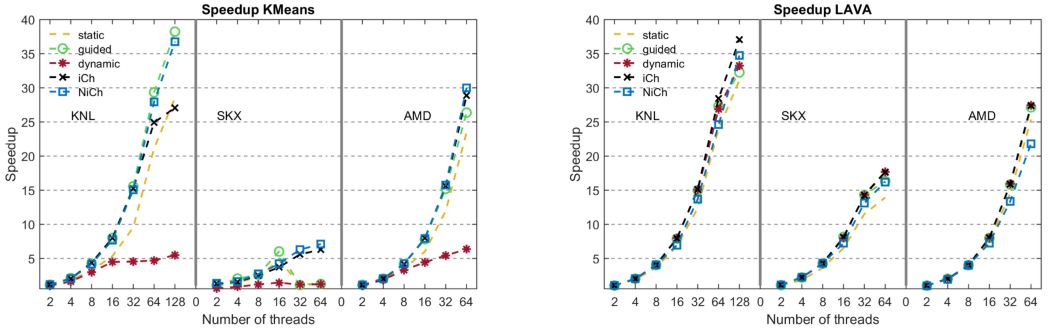
application. Additionally, we start to see a case where *dynamic* (for KNL, SKX, and SKX) and static (for SKX and AMD) are slightly worse.

Figure 3 presents the speedup performance on the BFS application with graphs generated with the number of edges from two distributions (i.e., BFS Uni and BFS Pow). This application scaled very well on KNL for *iCh*, *NiCh*, and *guided*. For BFS Uni, the off graph values for KNL are (*guided* {63.3, 102.2, 70.1}), (*iCh* {60.5, 91.3, 61.1}), and (*NiCh* {61.5, 92.7, 71.2}). Therefore, *guided*, *iCh*, and *NiCh* do about the same in terms of performance on KNL (with *NiCh* doing slightly worse than the other two). This is not the same story for SKX and AMD. For these two systems, *NiCh* does slightly better on SKX and *guided* does slightly better on AMD. Both *NiCh* and *guided* do better than *iCh* at the endpoint. When the distribution of edges becomes less uniform (BFS Pow), the trend with *guided*, *iCh*, and *NiCh* is about the same except that *NiCh* does not do as well on SKX. The off graph KNL values for BFS Pow are (*guided* {77.9, 132.9, 93.8}), (*iCh* {74.6, 117.2, 84.5}), and (*NiCh* {75.24, 117.2, 88.7}). The reason that static and dynamic scheduling do so poorly is because of KNL's architecture. The lack of a low-latency L3 cache and the poor single-core performance mean that schedulers with a lot of inter-thread communication (like *dynamic*) or schedulers that result in low cache coherence (like *static*) perform very poorly on this system. Because *NiCh* does so well on KNL, we are therefore able to deduce that it provides better cache coherence and less inter-thread communication than *static* or *dynamic* at the tested chunk sizes.



(a) BFS Uni (b) BFS Pow

Fig. 3. Speedup for the BFS application. We note that for SKX and AMD, *dynamic* may have a speedup < 1. For BFS, *NiCh* allows scaling past the point of one thread per core that limits *iCh* on SKX and AMD.



(a) Kmeans (b) LavaMD

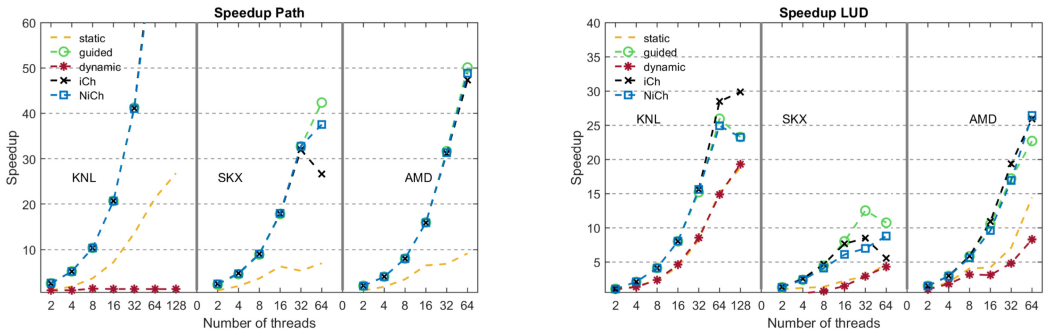
Fig. 4. Speedup for Kmeans and LavaMD applications. For Kmeans, *NiCh* allows better scaling than *iCh* past the point of one thread per core. However, LavaMD does not have the same impact, and simpler methods like *static* and *iCh* are better than *NiCh*.

The speedup results for Kmeans and LavaMD are presented in Figure 4. For Kmeans (see Figure 4(a)), we notice very different performance scaling among KNL, SKX, and AMD. Although most methods (besides *dynamic*) scale well on KNL and AMD, SKX does not scale well and *guided* does not scale past 16 threads. *iCh* and *NiCh* are able to at least continue to maintain their speedup for thread counts over 16 on SKX. We notice a slightly similar trend for LavaMD in which the application does not scale as well on SKX. However, in the case of LavaMD, all LS methods do about equally well, with *iCh* being slightly better at the endpoint.

Figure 5 presents the speedups for the Path and LUD applications. The Path application scales well on KNL with the speedup going off the graph. The values off the graph are (*guided* {80.9, 126.4}), (*iCh* {80.6, 121.5}), and (*NiCh* {78.5, 116.9}). We notice that *guided*, *iCh*, and *NiCh* are the clear choices for Path. For the LUD application, again the top three are *guided*, *iCh*, and *NiCh*. This performance of LUD demonstrates our conjecture that even LUD (which may be considered a regular application by many) can still be handled by *iCh* and *NiCh*.

6.1 Discussion

Overall, we can conclude that *guided*, *iCh*, and *NiCh* are all likely good choices. The original work for *iCh* [4] demonstrated that *iCh* may be a better choice than *guided* because of applications such



(a) Path (dynamic < 1) (b) LUD

Fig. 5. Speedup for Path and LUD applications. For Path, *NiCh* is normally better than *iCh*. For LUD, the more regular nature of the application generally allows *iCh* to do better than *NiCh* for KNL.

as Syn Exp Dec. In these cases, *guided* does not scale well, but *iCh* seems not to be impacted nearly as significantly. However, this research poses the following question: is *NiCh* necessary for modern systems? We believe the answer to be *maybe*. For systems like KNL and SKX, *iCh* does as well as *NiCh* except in several cases where the number of threads is greater than the number of cores (e.g., BFS Uni, Kmeans, and Path). The other main place where *iCh* does not seem to scale as well is when fully utilizing AMD, which is arguably the system architecture with the highest NUMA impact. However, *NiCh* does seem to also be negatively impacted by some distributions like *guided* (e.g., Syn Exp Dec). Therefore, the general recommendation from our observations is to continue to utilize *iCh* in cases where we do not have a good understanding of the workload distribution and not trying to fully utilize the whole processor. Despite this recommendation, the merit of *NiCh* can be seen in AMD with better on more extreme NUMA cases and when the thread count is higher than the number of cores. From this observation, we project that the merit of *NiCh* will improve as more architectures follow this design direction.

In the short term, the question arises if any of the adaptations of *NiCh* (e.g., variable chunk size, NUMA located deques, and NUMA distance-based stealing) would benefit other LS methods and other architectures. Our primary test system was AMD Milan due to its chiplet structure; however, there are many other systems that are following this pattern, as we pointed out previously. We believe that these systems, along with upcoming heterogeneous systems, will benefit the most. As such, we plan to continue to test *NiCh* on emerging systems, whereas testing the integration of the exact adaptation will depend on the method and architecture. An example of this is the shared NUMA located deques. Although they work well for our system and we show in Section 7.2 the sensitively to the number of threads sharing, more classical task-based work-stealing has shown that individual deques may still be ideal for the NUMA environment if implemented correctly [8] (i.e., which includes random stealing with NUMA bias). We do believe that there may be room to reconsider common traditional LS methods like factoring and trapezoid that have fallen from favor due to overheads with distributed queues and even factoring ratios that are adaptive on runtime information.

7 Sensitivity

The use of *NiCh* introduces three user parameters. These parameters are as follows. The parameter ϵ is used as a percentage of the range around the mean for categorizing low, normal, and high. The parameter p_q determines the number of threads per shared deque. Last, the parameter $KUpdate$ determines the number of chunks to be completed before the chunk size is updated.

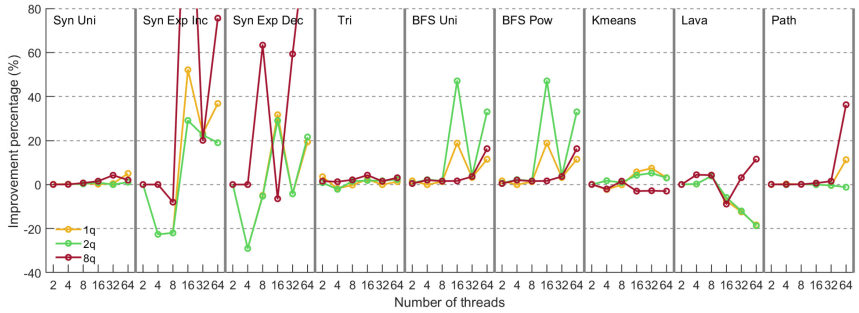


Fig. 6. Deque sensitivity (p_q). In most cases, using four threads per queue is better. Only in a couple of cases (i.e., the negative values) does a different number of threads per queue provide better performance.

7.1 Interval around the Mean ϵ

This parameter is also shared with *iCh*, and a detailed study of the sensitivity is provided in the original work [4], where we observe that there is little sensitivity of ϵ when ϵ is selected from the set of $\{.25, .33, .5\}$. We see in this study the same pattern that ϵ did not have a large impact for both *iCh* and *NiCh* when selected from the same set of values. Therefore, we continue to recommend simply utilizing $\epsilon = .33$.

7.2 Deque Size p_q

To gauge the impact of p_q, we consider how adjusting the number of threads per deque would affect the performance of *NiCh* on Milan. The logic of why we selected four threads per queue is stated in the algorithm. Therefore, we will consider the base performance to be the implementation that utilizes four cores per queue (i.e., 4_q). We define the metric for comparison as

$$p_q\_sensitivity(app, p_q, p) = \frac{time(app, p_q, p) - time(app, 4_q, p)}{time(app, 4_q, p)} \times 100, \tag{11}$$

where $time(app, p_q, p)$ is the time for running with *NiCh* on the benchmark application (*app*) parallelized with p threads and utilizing a shared queue of size p_q. Therefore, percentages greater than zero are times that are worse, and percentages less than zero are times that are better. Figure 6 presents the sensitivity of the deque for our benchmark applications in comparing the base size of four threads per deque to $\{1_q, 2_q, 8_q\}$. We note that we do not include `LUD` in this comparison because runtimes for this benchmark are very large and different deque sizes can make this too large to complete in a reasonable amount of time. Overall, the trend demonstrates that deque sizes other than 4_q normally decrease the performance. In most cases, this decrease is only small and on the order of the noise (e.g., `Syn Uni`, `Tri`, and `Kmeans`). However, there are a couple of benchmarks whose performance is greatly improved by the selection of the queue size (e.g., `Syn Exp Dec`, `Syn Exp Inc`, `BFS Uni`, `BFS Pow`, and `Path`). There exist only a couple of places where a different deque size resulted in better performance. In these cases, it is normally the reduction of two threads per deque that seems to increase performance marginally (e.g., `Syn Exp Dec`, `Syn Exp Inc`, and `LavaMD`).

7.3 Delay in Update *KUpdate*

Next, we study the sensitivity related to how often the chunk size should be updated. In *NiCh*, we suggest that the chunk size be updated after four accesses to the shared queue (K4), unlike *iCh* which updates every access of the queue. This suggestion is due to the overhead of utilizing more

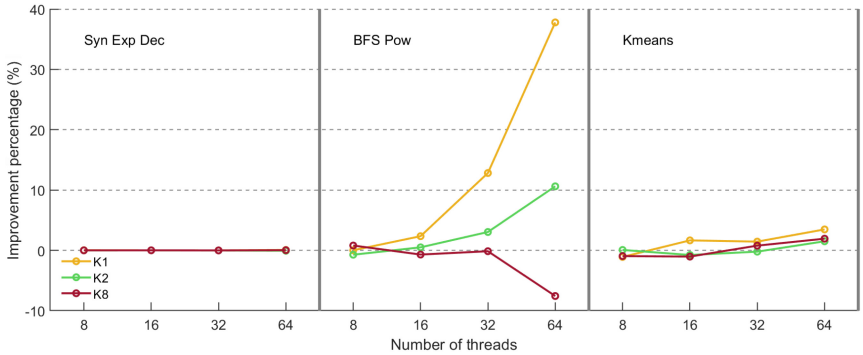


Fig. 7. Update sensitivity (*KUpdate*). In only a very small number of cases (i.e., negative values for `BFS Pow`), updating the chunk size less often than after four accesses (i.e., $K4$) improves performance. In most cases, either the number does not matter (e.g., `Syn Exp Dec` or low thread counts for `Kmeans`) or $K4$ is ideal.

threads per queue and not wanting the pace of one thread to dominate the changes to the shared chunk size. We define the sensitivity in a similar manner as with queue size—that is,

$$KUpdate\_sensitivity(app, K, p) = \frac{time(app, K, p) - time(app, K4, p)}{time(app, K4, p)} \times 100, \tag{12}$$

where $time(app, K, p)$ is the time for running with *NiCh* on the benchmark (*app*) parallized with p and utilizing K accesses to the shared queue before chunks size update. Again, a percentage greater than zero is the times when the particular mixture is worse, and a percentage less than zero is the times when the particular mixture is better. For our experiments, we fix $p_q = 4$ and only consider using more than eight threads (as fewer than eight threads would not really have much of an impact). Figure 7 presents the *KUpdate_sensitivity* for the benchmarks of `Syn Exp Dec`, `BFS Pow`, and `Kmeans`. We only present the data for these three because they are the benchmarks where *NiCh* is the most interesting. For `Syn Exp Dec`, we observe that the choice of update does not matter. Similarly, `Kmeans` is not very sensitive to the update choice, although K4 is overall better than the other options. For `BFS Pow`, the choice of the update matters with K1 and K2 being up to ~37% and ~11% worse. The only case where a different choice is significantly better (i.e., >5%) is on `BFS Pow` at 64 cores where K8 is ~8% better.

7.4 Number of Updates and Sensitivity to DVFS

To better understand the irregularity and how that impacts the adaptation of chunk size, we consider the number of times the chunk size is updated for each application on AMD. Table 2 provides the mean, variance, and range for the number of times the chunk size is updated across all shared deques. We provide the values for when the frequency is fixed to 2.0 GHz with `cpupower` (i.e., Fixed) and running with standard dynamic frequency scaling. We provide this number for both 32 cores and 64 cores. We notice that less irregular applications (e.g., `Syn Uni`) have much fewer updates than more irregular applications (e.g., `Tri`). However, this observation does not hold for all application and thread counts. An example of this inconsistency is `Path`, which ends up being very regular (see Table 1) but has as many updates to chunk size as `Kmeans` on 64 cores. We also notice, even though this is not reflected in the table, that a higher choice of ϵ results in fewer chunk size updates. This is expected behavior, as a higher ϵ implies that a greater deviation from the mean is necessary before a chunk size update occurs.

Table 2. Number of Updates with Fixed (Fixed) Frequency and Dynamic Frequency Scaling

	32 Cores			64 Cores		
Application	**Mean**	**Variance**	**Range**	**Mean**	**Variance**	**Range**
Syn Uni Fixed	0.0E+00	0.0E+00	0.0E+00	5.0E-01	2.7E-01	1.0E+00
Syn Uni	8.8E-01	1.3E-01	1.0E+00	8.8E-01	3.8E-01	2.0E+00
Syn Exp Inc Fixed	3.1E+01	8.8E+02	5.8E+01	2.7E+01	7.7E+02	5.8E+01
Syn Exp Inc	3.1E+01	8.3E+02	5.5E+01	2.8E+01	7.9E+02	5.9E+01
Syn Exp Dec Fixed	3.1E+01	9.3E+02	6.1E+01	3.1E+01	8.3E+02	6.3E+01
Syn Exp Dec	3.8E+01	8.7E+02	6.1E+01	3.1E+01	8.2E+02	5.9E+01
Tri Fixed	7.0E+01	7.4E+01	2.8E+01	8.5E+01	4.5E+03	2.3E+02
Tri	7.0E+01	7.4E+01	2.8E+01	8.5E+01	4.5E+03	2.3E+02
BFS Uni Fixed	2.4E+01	5.7E+02	5.4E+01	1.5E+02	1.9E+04	4.1E+02
BFS Uni	2.4E+01	5.7E+02	5.4E+01	1.5E+02	1.9E+04	4.1E+02
BFS Pow Fixed	1.4E+01	8.6E+01	2.4E+01	1.6E+02	2.2E+04	3.7E+02
BFS Pow	1.4E+01	8.6E+01	2.4E+01	1.6E+02	2.2E+04	3.7E+02
Kmeans Fixed	2.7E+01	1.1E+03	8.9E+01	2.1E+01	9.9E+02	1.1E+02
Kmeans	2.7E+01	1.1E+03	8.9E+01	2.1E+01	9.9E+02	1.1E+02
LavaMD Fixed	1.6E+00	1.2E+01	1.0E+01	6.4E+00	2.6E+02	4.8E+01
LavaMD	1.6E+00	1.2E+01	1.0E+01	6.4E+00	2.6E+02	4.8E+01
Path Fixed	3.3E+00	8.5E+00	8.0E+00	3.4E+01	2.9E+03	2.2E+02
Path	3.3E+00	8.5E+00	8.0E+00	3.4E+01	2.9E+03	2.2E+02
LUD Fixed	3.8E+02	7.1E+04	7.7E+02	5.1E+02	2.7E+05	1.7E+03
LUD	3.8E+02	7.1E+04	7.7E+02	5.1E+02	2.7E+05	1.7E+03

In terms of fixing the frequency, we notice that the performance scaling is about the same and therefore we do not include the raw scaling study. A more detailed way of studying the effects is based on the changes in the number of updates. Ideally, we would not want the number of updates to increase dramatically with the introduction of a dynamic frequency, as this would increase overhead. At the same time, we would want to see a small number of changes to compensate for the dynamic frequency. What we observe is that the number of updates is about the same. There is a very small change in the mean (i.e., <1%). Normally, this change is a small increase in the number of updates for the dynamic case (i.e., Syn Uni).

8 Related Work

There have been numerous LS methods throughout the years proposed because of the importance of finding one that works on a wide range of applications with little to no user input. *NiCh* sits at the intersection of three design techniques: work-stealing, non-fixed chunk size, and NUMA-aware methods.

In regard to work-stealing, one of the most popular work-stealing loop schedulers is implemented in Cilk [12]. Both *iCh* and *NiCh* use the THE protocol from Cilk-5, which allows for lockless work-stealing by observing if a queue has been modified mid-steal and rolling back changes without corrupting data. Cilk uses task-based scheduling, which is common among work-stealing schedulers. For example, the .NET Task Parallel Library (TPL) applies a similar approach to parallelism [18]. In regard to non-fixed chunk size, most work fits into two categories. The first is simple decreasing methods, such as *guided*, factoring [1, 13], and trapezoidal [23], that are not really learned from the input but are generic enough that they work with most inputs. These have trailed off in recent years, as *guided* has become the *de facto* standard. The second is those that base

their chunk size on some pre-runtime analysis of the code and workload. This category has been more recently studied as the penalties for imbalance are higher due to increasing core frequency relative to the cost of memory miss penalties. Recent work related to this includes BinLPT [22] and COWS [19]. BinLPT takes a direct approach to analyzing sample inputs and abstracting a schedule of chunk size. This works very well on simulation data where the sample can abstracted out in a meaningful way from the input but fails on more complex data inputs [4]. However, COWS uses a more complex method that analyzes the code at compile time and uses dynamic information at runtime to modify the method. This dynamic information is the number of iterations remaining that is stored in a very cache-friendly data structure. Although COWS does well on some applications, it does not do well on all their benchmarks, and in particular, they conclude that "cyclic" (i.e., *static* with a chunk size of 1, which is also tested in this work) is the best method overall.

There has been a lot of work related to NUMA-aware methods for particular applications [11] and with complex runtimes [2, 12]. However, there have been few that have tried to implement NUMA-aware methods into simple LS like OpenMP. Kappi [9] explored the concept of NUMA-aware thread stealing more than two decades ago in a very primitive way (i.e., a hierarchy stealing of those within the same socket and those across sockets). The result of this work inspired *NiCh* stealing even though Kappi's improvements were only marginal due to the architectures of the time.

9 Conclusion

In this work, we proposed the question of if the NUMA effects of more modern shared memory processors require a new NUMA-aware approach to LS. We constructed a new NUMA-aware LS method (*NiCh*) based on *iCh*, which is an adaptive chunk size LS method that has been demonstrated to provide near-optimal performance with little user input and tuning. We tested *NiCh* against *iCh* and other standard OpenMP LS methods on three different systems. We conclude that *NiCh* does have merit for systems with more extreme NUMA effects if the number of threads utilized is close to or over the number of cores. However, *NiCh* comes with the negative impact of not being as universally good as *iCh* (e.g., the performance on the `Syn Exp Dec` application) and not being as idiot-proof (i.e., it requires addition sensitivity parameters that are most likely hardware. although not application, dependent).

References

[1] Ioana Banicescu, Vijay Velusamy, and Johnny Devaprasad. 2003. On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. *Cluster Computing* 6, 3 (2003), 215–226.

[2] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC '12)*. 1–11. https://doi.org/10.1109/SC.2012.71

[3] Timothy J. Boerner, Stephen Deems, Thomas R. Furlani, Shelley L. Knuth, and John Towns. 2023. Access: Advancing innovation: NSF's advanced cyberinfrastructure coordination ecosystem: Services & support. In *Proceedings of the Conference on Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good (PEARC '23)*. 173–176.

[4] Joshua Dennis Booth and Phillip Allen Lane. 2022. An adaptive self-scheduling loop scheduler. *Concurrency and Computation: Practice and Experience* 34, 6 (2022), e6750. https://doi.org/10.1002/cpe.6750

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC '09)*. IEEE, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[7] Ian Cutress and Andrei Frumusanu. 2021. AMD 3rd Gen EPYC Milan Review: A Peak vs Per Core Performance Balance. Retrieved July 30, 2024 from https://www.anandtech.com/show/16529/amd-epyc-milan-review/4

[8] Justin Deters, Jiaye Wu, Yifan Xu, and I-Ting Angelina Lee. 2018. A NUMA-aware provably-efficient task-parallel platform based on the work-first principle. In *Proceedings of the 2018 IEEE International Symposium on Workload Characterization (IISWC '18)*. IEEE, 59–70.

[9] Marie Durand, François Broquedis, Thierry Gautier, and Bruno Raffin. 2013. An efficient OpenMP loop scheduler for irregular applications on large-scale NUMA machines. In *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, Berlin, Germany, 141–155. https://doi.org/10.1007/978-3-642-40698-011

[10] ECP. 2023. ECP Proxy Application, miniTri. Retrieved July 30, 2024 from https://proxyapps.exascaleproject.org/app/minitri/

[11] Michael Frasca, Kamesh Madduri, and Padma Raghavan. 2012. NUMA-aware graph mining techniques for performance and energy efficiency. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '12)*. IEEE. https://doi.org/10.1109/sc.2012.81

[12] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. ACM, New York, NY, USA. https://doi.org/10.1145/277650.277725

[13] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. 1992. Factoring: A method for scheduling parallel loops. *Communications of the ACM* 35, 8 (Aug. 1992), 90–101. https://doi.org/10.1145/135226.135232

[14] Humayun Kabir, Joshua Dennis Booth, and Padma Raghavan. 2014. A multilevel compressed sparse row format for efficient sparse computations on multicore processors. In *Proceedings of the 2014 21st International Conference on High Performance Computing (HiPC '14)*. 1–10. https://doi.org/10.1109/HiPC.2014.7116882

[15] Franziska Kasielke, Ronny Tschüter, Christian Iwainsky, Markus Velten, Florina M. Ciorba, and Ioana Banicescu. 2019. Exploring loop scheduling enhancements in OpenMP: An LLVM case study. In *Proceedings of the 2019 18th International Symposium on Parallel and Distributed Computing (ISPDC '19)*. 131–138. https://doi.org/10.1109/ISPDC.2019.00026

[16] A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos. 2006. History-aware self-scheduling. In *Proceedings of the 2006 International Conference on Parallel Processing (ICPP '06)*. ACM, New York, NY, USA, 185–192.

[17] Henning Meyerhenke. 2023. DIMACS. Retrieved July 30, 2024 from https://www.cc.gatech.edu/dimacs10/index.shtml

[18] Microsoft Learn. 2023. TaskScheduler Class. Retrieved July 30, 2024 from https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskscheduler

[19] Prasoon Mishra and V. Krishna Nandivada. 2024. COWS for high performance: Cost aware work stealing for irregular parallel loops. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024), Article 12, 26 pages. https://doi.org/10.1145/3633331

[20] Nevine Nassif, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. 2022. Sapphire Rapids: The next-generation Intel Xeon scalable processor. In *Proceedings of the 2022 IEEE International Solid-State Circuits Conference (ISSCC '22)*, Vol. 65. 44–46. https://doi.org/10.1109/ISSCC42614.2022.9731107

[21] Mantevo Organization. 2023. Mantevo Project. Retrieved July 30, 2024 from https://mantevo.github.io/

[22] Pedro Henrique Penna, Antônio Tadeu A. Gomes, Márcio Castro, Patricia D. M. Plentz, Henrique C. Freitas, Francois Broquedis, and Jean-François Méhaut. 2019. A comprehensive performance evaluation of the BinLPT workload-aware loop scheduler. *Concurrency and Computation: Practice and Experience* 31, 18 (2019), e5170.

[23] T. H. Tzen and L. M. Ni. 1993. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems* 4, 1 (1993), 87–98. https://doi.org/10.1109/71.205655

[24] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. 2022. Memory performance of AMD EPYC Rome and Intel Cascade Lake SP server processors. In *Proceedings of the 13th ACM/SPEC International Conference on Performance Engineering (ICPE '22)*. ACM, New York, NY, USA, 165–175. https://doi.org/10.1145/3489525.3511689

[25] B. P. Welford. 1962. Note on a method for calculating corrected sums of squares and products. *Technometrics* 4 (1962), 419–420.

Received 2 December 2023; revised 1 May 2024; accepted 15 July 2024