NextGen-Malloc: Giving Memory Allocator Its Own Room in the House

Ruihao Li  
liruihao@utexas.edu  
The University of Texas at Austin

Qinzhe Wu  
qw2699@utexas.edu  
The University of Texas at Austin

Krishna Kavi  
Krishna.Kavi@unt.edu  
University of North Texas

Gayatri Mehta  
Gayatri.Mehta@unt.edu  
University of North Texas

Neeraja J. Yadwadkar  
neareraja@austin.utexas.edu  
The University of Texas at Austin and VMware Research

Lizy K. John  
ljohn@ece.utexas.edu  
The University of Texas at Austin

ABSTRACT

Memory allocation and management have a significant impact on performance and energy of modern applications. We observe that performance can vary by as much as 72% in some applications based on which memory allocator is used. Many current allocators are multi-threaded to support concurrent allocation requests from different threads. However, such multi-threading comes at the cost of maintaining complex metadata that is tightly coupled and intertwined with user data. When memory management functions and other user programs run on the same core, the metadata used by management functions may pollute the processor caches and other resources.

In this paper, we make a case for offloading memory allocation (and other similar management functions) from main processing cores to other processing units to boost performance, reduce energy consumption, and customize services to specific applications or application domains. To offload these multi-threaded fine-granularity functions, we propose to decouple the metadata of these functions from the rest of application data to reduce the overhead of inter-thread metadata synchronization. We draw attention to the following key questions to realize this opportunity: (a) What are the tradeoffs and challenges in offloading memory allocation to a dedicated core? (b) Should we use general-purpose cores or special-purpose cores for executing critical system management functions? (c) Can this methodology apply to heterogeneous systems (e.g., with GPUs, accelerators) and other service functions as well?

CCS CONCEPTS

• Software and its engineering → Memory management.

KEYWORDS

Memory Management

ACM Reference Format:

1 INTRODUCTION

Data intensive applications, including machine learning (ML) training and inference serving rely on efficient management of large amounts of data. With these increasing capacities and deep memory hierarchies, the memory wall [35] (viz., the gap between processor speeds and memory access latencies) is becoming an ever increasing impediment. ML accelerators spend only 23% of the overall cycles on computations and more than 50% on data preparation [15]. Even in general purpose warehouse-scale datacenters, 50% of compute cycles are idle waiting for memory accesses [14, 16], causing performance degradation for the applications. To alleviate such performance bottlenecks, efficient memory management mechanisms [14, 18, 20] that reduce memory needs (by reducing fragmentation) and cache misses (by reducing conflicts between metadata and application data) are crucial.

To address the need for efficient memory allocation and management, multiple software solutions, in the form of optimized Malloc libraries, have been created (e.g., TCMalloc [14] from Google and Mimalloc [18] from Microsoft). Such libraries are typically multi-threaded to support concurrent requests for memory allocation and management, and rely on complex metadata to accomplish the allocation.
and management. However, if the same core executes memory management and application code, maintaining such metadata can pollute processor caches, cause conflicts on other resources and can affect overall performance.

To understand the effectiveness of existing software solutions for memory management, we compare the performance of four different memory allocators (PTMalloc2 [12], Jemalloc [9], TCMalloc [14], and Mimalloc [18]) for xalancbmk (a representative workload from SPEC cpu2017 [23, 25], which performs transformations on XML data). We observe that with an enhanced memory allocator, the overall system performance can be improved by as much as 1.72× (see Figure 1). On other allocation intensive workloads, e.g., xmalloc and cache-scratch in mimalloc-bench [18, 22], performance variance can be more than 10× depending on the allocator used. We notice that the choice of memory allocator not only impacts the performance of the allocator code itself but also other parts of the program. For example, although only 2% of the execution time is spent on malloc and free by xalancbmk, we can see as much as 72% difference in overall execution time when Mimalloc is used instead of PTMalloc2. This makes a compelling case for further investigating and optimizing these memory management functions.

Existing memory allocation mechanisms that are implemented in software achieve higher performance compared with the default Glibc (PTMalloc2) memory allocators. However, they still fall short in leveraging system-level optimizations, like reducing cache pollution and TLB misses, which cannot be implemented without understanding underlying hardware. In addition, software-only solutions always suffer from balancing allocation speed and memory consumption (partly due to fragmentation), making it difficult to develop a one-size-fits-all allocation solution.

Alternatively, hardware accelerators are considered for implementing memory allocation mechanisms. For instance, Kanev et al. [17], proposed a separate cache inside main CPUs for memory management algorithms, thus eliminating cache conflicts between memory management metadata and application data, while still utilizing powerful main CPUs for memory management. Mass et al. [19], used a near-memory accelerator to offload garbage collection functions. However, most hardware solutions rely on customized hardware units, that (a) make it infeasible to use them as general purpose memory management solution, or (b) necessitate frequent changes to the hardware as algorithms evolve.

Next generation memory allocators will likely continue to be complex. One way to handle their complexity without impacting application performance is to isolate the allocation functions from rest of the code and provide separate resources to it. This will prevent allocators from polluting the cache and interfering with other metadata of applications. However, current allocators cannot easily be “plucked” out of the code and offloaded to dedicated cores, because the metadata is usually tightly coupled to the user data.

In this paper, we introduce NextGen-Malloc, a novel memory allocator that restructures the memory management metadata and makes it possible to offload the allocation to a separate dedicated core. The design of NextGen-Malloc necessitates innovations in software, hardware and their codesign. Just like a baby in a big family, the memory allocator is growing up. Now it is time to give it a new room (core) in our house (CPU). New research questions will arise with the growing child NextGen-Malloc. We will explore these questions in this paper. How to tradeoff the overhead (additional inter-core communication) and the benefit (a reduction of cache pollution and asynchronous execution) of offloading memory allocators? The choice of the room type (viz., type of processing core) is another research question. Should the room be the same as other rooms (i.e., other CPU cores), or a small room is enough for memory allocation? Can the room be used for other functions instead of exclusively for memory allocation?

2 BACKGROUND AND MOTIVATION

This section identifies the primary reasons and inefficiencies contributing to the performance differences among different memory allocators.

2.1 Memory Allocators

Memory management operations exist at two levels: user level and kernel level. User-level memory management operations are implemented by a UMA (user-level memory allocator), such as PTMalloc2 [12] (the default Glibc implementation), TCMalloc [14] (by Google), and Mimalloc [18] (by Microsoft). There are other domain-specific UMAs as well, such as NVAlloc [7] for non-volatile memory. For the kernel-level operations, mmap() system calls are used to manage private anonymous mapping segments for each process. But relying on mmap() system call for each malloc() can result in...
Table 1: Processor performance monitor data for `xalanckm`. TLB misses vary more than 10x and LLC load misses vary 4x between TCMalloc and PTMalloc2.

<table>
<thead>
<tr>
<th>Allocator</th>
<th>PTMalloc2</th>
<th>JeMalloc</th>
<th>TCMalloc</th>
<th>Mimalloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycles</td>
<td>1.177E+12</td>
<td>7.115E+11</td>
<td>7.091E+11</td>
<td>6.959E+11</td>
</tr>
<tr>
<td>instructions</td>
<td>1.282E+12</td>
<td>1.320E+12</td>
<td>1.254E+08</td>
<td>1.254E+08</td>
</tr>
<tr>
<td>LLC-load-misses</td>
<td>4.059E+08</td>
<td>9.445E+07</td>
<td>1.016E+08</td>
<td>1.477E+08</td>
</tr>
<tr>
<td>LLC-store-misses</td>
<td>3.554E+08</td>
<td>1.630E+08</td>
<td>1.321E+08</td>
<td>1.321E+08</td>
</tr>
<tr>
<td>dTLB-load-misses</td>
<td>1.804E+09</td>
<td>1.482E+08</td>
<td>1.641E+08</td>
<td>1.628E+08</td>
</tr>
<tr>
<td>dTLB-store-misses</td>
<td>3.669E+07</td>
<td>2.937E+07</td>
<td>2.591E+07</td>
<td>2.787E+07</td>
</tr>
</tbody>
</table>

in significant performance penalties caused by switching between user and kernel modes. To minimize these overheads, most UMAs request an entire page\(^1\), which is usually larger than the size requested by `malloc()` calls. For subsequent `malloc()` calls, the UMA can satisfy the requests out of the page and make additional `mmmap()` calls only when space in the allocated page is exhausted.

**Observation:** Managing the memory space within pre-allocated pages in a way that strikes a balance between allocation speed and memory fragmentation is an open research problem.

### 2.2 Malloc impacts more than you think!

The implementation of memory management functions can potentially have significant impact on the overall execution time of applications. In this section, we present evidence to support the claim we made earlier: “Though only 2% of time is spent on memory allocation, more than 70% performance difference can exist with different implementations”.

The above claim may appear counter-intuitive. Table 1 lists the hardware PMU (Performance Monitor Unit) event counts for `xalanckm` with the four memory allocators. Based on the profiling results, the number of LLC (Last Level Cache) load and store misses, and the number of dTLB (data Translation Lookaside Buffer) load misses are reduced dramatically (besides absolute miss numbers, the miss rate reduced as well) on the three state-of-the-art industry-level allocators (Jemalloc, Mimalloc, and TCMalloc) compared with the default Glibc allocator (PTMalloc2). The impact of TLB misses is non-negligible, which can incur 100s of cycles in modern processors [11, 27]. In addition, in warehouse-scale computers, half of the CPU back-end cycles can be spent on serving data cache requests [16].

**Observation:** TLB and LLC misses are key factors affecting the overall performance and sensitive to different memory allocators. Alleviating the cache pollution is a potential direction for the next-generation memory allocators to explore, to achieve a lower number of TLB and LLC misses.

---

\(^1\)The page size for UMA may be different from OS page size [14].

---

Table 2: PMU data for `xmalloc` on TCMalloc using different number of threads. LLC misses increase more than 10x when the number of threads increases from 1 to 8.

<table>
<thead>
<tr>
<th># of threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycles</td>
<td>4.333E+10</td>
<td>6.708E+10</td>
<td>1.148E+11</td>
<td>1.963E+11</td>
</tr>
<tr>
<td>instructions</td>
<td>2.387E+10</td>
<td>2.979E+10</td>
<td>3.893E+10</td>
<td>4.874E+10</td>
</tr>
<tr>
<td>LLC-load-misses</td>
<td>1.223E+05</td>
<td>2.196E+05</td>
<td>2.477E+06</td>
<td>5.402E+07</td>
</tr>
<tr>
<td>LLC-store-misses</td>
<td>3.676E+06</td>
<td>4.213E+06</td>
<td>1.650E+07</td>
<td>5.402E+07</td>
</tr>
</tbody>
</table>

### 2.3 Existing Allocators are Limiting

Modern UMAs support concurrent memory `malloc()` and `free()` operations for multi-threaded programs, which requires processing `malloc()` and `free()` requests from different physical cores and different address spaces simultaneously. As the number of cores within a single socket keeps increasing, it will compound the challenges for software UMAs in handling thread contentions: the inter-core metadata synchronization requires atomic operations (or locks) to maintain the upper-level global free lists. Software mutex locks are used to control access to metadata to process requests from different cores. The cost of using such software locks is high since cross-core communication is involved, causing a critical performance bottleneck as the communication overheads increase with the number of cores [29, 34].

Multiple software solutions have been used to address the multi-thread contention issue. TCMalloc [13] uses per-CPU/thread cache to maintain metadata associated with each logical core, avoiding locks for most memory allocations and deallocations. Mimalloc [18] uses three page-local shared free lists to increase locality, avoid contention, and support a highly-tuned allocation and free on fast path. However, maintaining thread-local caches will increase metadata size, resulting in more heap memory consumption and more cache pollution for the user program. Also, the number of logical threads may exceed the number of available physical cores, resulting in more overhead in switching between thread-local metadata.

To demonstrate the potential cache pollution in multi-threaded UMAs, we collect the PMU event counts for `xmalloc` [4] workload\(^2\) on TCMalloc (listed in Table 2). As the number of threads increases, they contend with each other for accessing the thread-local cache. This results in a significant LLC miss increment (more than 10x when the number of threads increases from 1 to 8).

**Observation:** It is challenging to address thread contention well without understanding the underlying hardware mechanisms for synchronization and communication.

\(^2\)`xmalloc` [4] is a multi-threaded benchmark by Lever and Boreham and used to exercise cases where a thread allocates data but a different thread deallocates the allocated blocks.
**Expectation:** If the execution of the memory allocator and user program are separated, all thread-local metadata can reside within one core, alleviating cache pollution. This can be achieved by making `malloc()` run on a dedicated core separating from other user programs.

## 3 OPPORTUNITIES

Merely offloading memory allocators to a dedicated core is not what NextGen-Malloc entails. There are several challenges to be solved to make it a reality and to achieve its full potential. We present some impactful avenues here.

### 3.1 Offload Malloc to its Own Core

By offloading memory management functions to a dedicated core, the overall performance can potentially benefit from parallel/concurrent execution and a reduction of cache pollution. However, offloading these fine granularity functions is still a challenge for current computer systems, considering the overhead of inter-core communications and the tight coupling between data and metadata. We detail the challenges and propose remedies that can make NextGen-Malloc a beneficial reality.

#### 3.1.1 Challenges

While it is not uncommon to see CPU resource underutilization in datacenters (for example, 60% of the Virtual Machines on Microsoft Azure have an average CPU utilization lower than 20% [6, 33]), and while offloading system functions to free cores to boost overall performance has been tried for other problems (e.g., Shenango [24] dedicates a single busy-spinning core per machine to a centralized software entity), there are unique challenges to offloading memory allocators. The majority of memory allocation function calls are fine-grained and can be finished within 100 cycles [17]. Thus memory allocation amounts to 0.05 μs on a 2GHz machine, while in Shenango [24] the granularity is 5 μs (the busy-spinning core reschedules among cores every 5 μs, and the duration of each job is usually longer than 5 μs). In comparison to allocation time-scales, the overhead of inter-core communication is non-negligible. A single Atomic Read-Modify-Write (RMW) instruction needed for inter-core synchronization can take 67 cycles on average on a Sandy Bridge machine [26], and almost 700 cycles in the worst case [3]. MMT [31] explored offloading memory allocation tasks to a memory management thread in a 2-core system. The performance did not improve without aggressive preallocations, which makes it only work for workloads with known allocation patterns.

**Expectation:** Whether the overall performance can benefit from offloading memory allocators to a dedicated core is still an open question. It depends on whether the inter-core synchronization overhead can be reduced to acceptable levels.

#### 3.1.2 Strategy 1 – Decoupling

Decoupling the metadata memory space from user data is one of the solutions that may lead to the overall performance gains from offloading the memory allocator. Although MMT [31] offloads the memory allocator to a separate thread, the cache pollution issue cannot be alleviated unless all metadata is accessed exclusively by the dedicated core for memory management. Before discussing allocator decoupling, we will discuss Metadata Layout first. Figure 2 shows two representative metadata layouts, Aggregated Layout (used by Mimalloc) and Segregated Layout (used by TCMalloc).

In Aggregated Layout, the first 8 bytes (assuming 64-bit word size) of each free block are used as the pointer to the next free block. Global Head and Tail pointers are used for each page (each page has a different number of blocks, depending on the block size (the block size is not necessarily a power of 2 [13, 18]). If a block is accessed directly after the `malloc()` call, since 56 bytes (assuming 64-byte cache line size) of the allocated data will already be cached by the `malloc()` function, better spatial localities will result for the user program. In addition, the memory block can easily be located in the `free()` phase, since each block can be indexed by the address directly.

However, optimizing locality for aggregated layout calls for (1) the allocator and user program must be executed on the same physical core, (2) the allocated memory must be accessed immediately after it is allocated, (3) the cached memory block must not be evicted before it is deallocated.

In Segregated Layout, metadata is decoupled from the allocated storage, which may address the cache pollution issue. Instead of an 8-byte pointer, a smaller index (16-bit for example) can be used to indicate the location of the free page, reducing memory fragmentation as well. Segregated layout requires more complicated control logic for locating the pointer to the freed data during the `free()` phase. Cache pollution can still exist since CPUs have to access data needed for malloc functions and other computational functions as well.

**Expectation:** Trade-offs always exist between the aggregated and segregated layouts and there is no global optimal layout. However, segregated layout is more suitable for offloading memory allocators in NextGen-Malloc. With segregated layout, the address space of metadata and user data can be separated. The additional cost in `free()` would not be an issue, since the entire `free()` phase is not on the critical path and can be executed asynchronously in the dedicated core.

#### 3.1.3 Strategy 2: Removing unnecessary atomic operations in UMAs

In current UMA designs, if one thread tries to free a memory block that was allocated by another running thread, contention will result (as described in Section 2.3) and atomic operations are required to sequentialize changes.
made to critical resources to avoid race conditions. When offloading the memory allocator to a dedicated core, all memory management operations will be handled by one specific core, metadata contention should be eliminated theoretically (from a hardware perspective). Therefore, we can remove redundant atomic operations in UMAs, since sequential execution can be guaranteed if all allocation codes are running in one specific core.

**Questions:** Whether the contention is reduced or not is still an open question, depending on the trade-off between the introduced synchronization overhead in NextGen-Malloc and removed atomic operations in current UMAs.

### 3.2 Type of Core to Offload to

One possibility is to offload allocation to an idle core in the system. An alternative solution is to use a near-memory core. Harvesting an idle core may be more cost-effective, while a near-memory core could be energy-efficient and/or offer opportunities for customization. A general-purpose core makes it compatible with other system instruction sets and ease of compiling UMAs. A single-threaded in-order integer CPU may be adequate for a near-memory design, since memory address computations do not need floating point operations or complex control logic. The near-memory core will likely have lower memory access latencies; thus requiring only a small (micro) cache for buffering metadata. It is crucial to explore these design choices as they have the potential to improve performance while reducing energy/power consumption of memory management functions.

### 3.3 What else can NextGen-Malloc offer?

In addition to UMAs, NextGen-Malloc can also be applied to other memory management scenarios. There are several opportunities that can be explored extending NextGen-Malloc.

#### 3.3.1 Optimizing GPU Malloc

In addition to CPU-only memory allocation systems, memory management functions on CPU-GPU heterogeneous systems can be included in NextGen-Malloc as well. When GPU memory spaces are included, more trade-offs will be there. For example, in Nvidia GPUs, UVM (Unified Virtual Memory) [1, 36] is used to make the address space shared across the CPU hosts and GPU devices. Redundant memory transmission, data allocation granularity, and address space mapping are open questions. Asynchronous allocation can be used, which can also be part of the asynchronous CUDA memory copy. Both CPU and GPU memory allocators can be decoupled from user programs for faster address translation. All these features can be explored in NextGen-Malloc.

#### 3.3.2 Other Functions to Offload

In addition to user-level memory allocations, NextGen-Malloc can be used for other memory management functions. This flexibility comes from the programmable feature of NextGen-Malloc. In the kernel space, NextGen-Malloc can be combined with other resource management mechanisms, e.g., Caladan [10], to balance CPU and memory usage of different tasks. More intelligence can be programmed to observe allocation requests and utilize such information to predictively preallocate memory to reduce allocation latencies.

Also, there are opportunities in memory management for other scenarios, including managed languages (e.g., Java and .NET) and serverless functions (e.g., AWS Lambda [2] and Azure Functions [21]). With GC (Garbage Collector) invoked, LLC miss rate is reduced in .NET [8]. With different GC settings, the performance of the program can be affected a lot. Research opportunities for using NextGen-Malloc to process garbage collection will be worth exploring. Booting a function in FaaS (Functions-as-a-Service) systems through cold start can introduce extensive overhead, including additional memory consumption and allocation time [5, 28, 30, 32]. To avoid duplicate runtime library initialization in different containers, NextGen-Malloc can be extended to monitor inter-process memory heap similarities in FaaS systems as well.

### 4 NEXTGEN-MALLOC IS FEASIBLE

As outlined in previous sections, NextGen-Malloc can support asynchronous execution and reduce LLC and TLB misses by separating the memory space of memory management
function metadata between user data, but at the cost of additional inter-core communication. In this section, we model this trade-off analytically and validate it by prototyping NextGen-Malloc for execution on real machines and comparing it with state-of-the-art memory allocators.

4.1 NextGen-Malloc Performance Estimates

We use xalancbmk as an example in an analytical model to demonstrate the potential performance improvement if NextGen-Malloc is used. In xalancbmk, malloc() and free() functions are called 279,759, 405 times in total (138, 401, 260 malloc() and 141, 394, 145 free()). The performance overhead for these functions comes from atomic operations needed at the beginning and end of each malloc and free function calls. Assuming a 67-cycle latency for one atomic operation [3], there will be around 75 billion additional cycles introduced by NextGen-Malloc. We assume that any performance benefit of NextGen-Malloc is from reduced LLC and TLB misses. Comparing Mimalloc to Glibc, we can calculate that the average LLC and TLB miss penalty is 214 cycles. To amortize the overhead, NextGen-Malloc has to achieve a reduction of at least 1.25 Cache/TLB misses in each malloc()/free() and the subsequent user program before the next malloc()/free() comes. This is possible to achieve in NextGen-Malloc, since there are 7 load and store instructions in each malloc() and 10 in each free() (for Mimalloc).

4.2 Validating NextGen-Malloc

We prototype NextGen-Malloc on an AWS-A1 bare metal machine, with 16 Armv8-A Cortex-A72 (ARM uses a weaker memory model, which reduces the inter-core synchronization overhead.) cores and 32 GB memory, by overwriting functions defined in standard C libraries, e.g., malloc() and free(), and compile it as a shared library. In NextGen-Malloc, we spawn a child thread from the main thread when the process is forked, pin it to a specific core, and let the spawned-thread check signals from the main thread for incoming malloc() and free() requests. We use the malloc() function as an example (shown in Code 1) to demonstrate the synchronization system in NextGen-Malloc, which is developed for the communication between the spawned child thread and main threads.

Two atomic variables malloc_ready and malloc_done are used at the beginning and end of nextgen_malloc() and malloc(), which are executed by the child thread and main thread, respectively. The requested_size and allocated_block are the input and output of malloc() functions, and this information is transferred between two threads. In this way, all other malloc() related metadata will be stored exclusively on the dedicated core, alleviating cache pollution that exists in state-of-the-art memory allocators.

```
void nextgen_malloc() { // executed in child thread
  while (true)
    if (malloc_ready)
      size_t size = atomic_load(requested_size);
      void *block = normal_malloc(size);
      atomic_store(allocated_block, block);
      atomic_store(malloc_done, 1);
      atomic_store(malloc_ready, 0);
  }
  void malloc(size_t size) { // executed in main thread
    while (atomic_load(malloc_ready));
    atomic_store(requested_size, size);
    atomic_store(malloc_ready, 1);
    while (!atomic_load(malloc_done));
    return atomic_load(allocated_block);
  }
```

Pseudo Code 1: NextGen-Malloc Synchronization.

We still use xalancbmk as the workload. We make a side-by-side comparison between Mimalloc and NextGen-Malloc in Table 3. NextGen-Malloc achieves a 4.51% improvement, which is also coming from a reduction of dTLB load, LLC load, and LLC store misses.

Table 3: Execution time, dTLB MPKI (misses per kilo instructions), LLC load MPKI improve by 4.5%, 43%, and 22% respectively on moving from Mimalloc to the proposed NextGen-Malloc. AWS-A1 (ARMv8) Platform was used for this study.

<table>
<thead>
<tr>
<th></th>
<th>Mimalloc</th>
<th>NextGen-Malloc</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td>532s</td>
<td>508s</td>
<td>4.51%</td>
</tr>
<tr>
<td>dTLB-load-MPKI</td>
<td>6.092</td>
<td>3.452</td>
<td>43.33%</td>
</tr>
<tr>
<td>LLC-load-MPKI</td>
<td>8.911</td>
<td>6.889</td>
<td>22.69%</td>
</tr>
<tr>
<td>LLC-store-MPKI</td>
<td>0.783</td>
<td>0.496</td>
<td>36.59%</td>
</tr>
</tbody>
</table>

5 CONCLUSION

In this paper, we draw attention to the opportunity to achieve significant performance gains by offloading fine-granularity service functions like memory allocation to a separate core/device. These memory allocation functions cause cache pollution when they are run on the same core as the application code due to the tight coupling of their metadata with user data. We discuss challenges in offloading such management functions to a separate core and propose avenues for future research.

Acknowledgement: We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their constructive feedback and insightful comments. This research was supported in part by NSF grant number 1763848 and 1828105, and gifts from Cisco and Amazon Web Services (AWS). The authors would also like to acknowledge the computing servers donated by Ampere Computing and TACC. Any opinions, findings, conclusions, or recommendations are those of the authors and not of the National Science Foundation or other sponsors.
REFERENCES


