Work Stealing and Persistence-based Load Balancers for Iterative Overdecomposed Applications

Jonathan Lifflander
University of Illinois
Urbana-Champaign
jliffl2@illinois.edu

Sriram Krishnamoorthy
Pacific Northwest National Lab
sriram@pnnl.gov

Laxmikant V. Kale
University of Illinois
Urbana-Champaign
kale@illinois.edu

ABSTRACT

Applications often involve iterative execution of identical or slowly evolving calculations. Such applications require incremental rebalancing to improve load balance across iterations. In this paper, we consider the design and evaluation of two distinct approaches to addressing this challenge: persistence-based load balancing and work stealing. The work to be performed is overdecomposed into *tasks*, enabling automatic rebalancing by the middleware. We present a hierarchical persistence-based rebalancing algorithm that performs localized incremental rebalancing. We also present an active-message-based *retentive* work stealing algorithm optimized for iterative applications on distributed memory machines. We demonstrate low overheads and high efficiencies on the full NERSC Hopper (146,400 cores) and ALCF Intrepid systems (163,840 cores), and on up to 128,000 cores on OLCF Titan.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; J.2 [**Physical Sciences and Engineering**]: Chemistry

Keywords

dynamic load balancing, work stealing, persistence, iterative applications, task scheduling, hierarchical load balancer

1. INTRODUCTION

Applications often involve iterative execution of identical or slowly evolving calculations. Many such applications exhibit significant complexity and runtime variation to preclude effective static load balancing, requiring incremental rebalancing to improve load balance over successive iterations.

A popular approach to addressing the load balancing challenge is *overdecomposition*. Rather than parallelize to a specific processor core count, the application-writer exposes parallelism by decomposing work into medium-grained tasks. Each task is coarse enough to enable efficient execution (tiling, vectorization, etc.) and potential migration, while being fine enough to expose significantly higher application-level parallelism than is required by the hardware. This allows the middleware to manage the mapping of the tasks to processor cores, and rebalance them as needed.

This paper focuses on balancing the computational load across processor cores for overdecomposed applications, where static or start-time approaches are insufficient in achieving effective load balance. Such applications require periodic rebalancing of the load to ensure continued efficiency.

Applications that retain the computation balance over iterations, with gradual change, are said to adhere to the *principle of persistence*. Persistence-based load balancers redistribute the work to be performed in a given iteration based on measured performance profiles from previous iterations. We present a hierarchical persistence-based load balancing algorithm that attempts to localize the rebalance operations and migration of tasks. The algorithm greedily rebalances "excess" load rather than attempting a globally optimal partition, which could potentially incur high space overheads.

Work stealing is an attractive alternative for applications with significant load imbalance within a phase, or applications with workloads that cannot be easily profiled. Work stealing ameliorates these problems by actively attempting to find work until termination of the phase is detected. This approach has been successfully employed in domains where the load imbalance cannot be computed a priori or varies significantly across consecutive invocations [19]. We present an active-message-based work stealing algorithm optimized for iterative applications on distributed memory machines. The algorithm minimizes the number of remote roundtrip latencies incurred, reduces the duration of locked operations, and takes into account data transfer time when stealing tasks across the communication network. *Retentive* work stealing augments this algorithm with knowledge of execution profiles from the previous iteration to enable incremental rebalancing.

The scalability and overheads incurred by these algorithms are evaluated using candidate benchmarks. We observe that the persistence-based load balancer produces effective load distributions with low overheads. We demonstrate work stealing at over an order of magnitude higher scale than prior published results. While more scalable than widely believed, work stealing does not perform as well as persistence-based load balancing. Retentive stealing, which borrows from the persistence-based load balancer to retain information from the previous iteration, is shown to adapt better to iterative

HPDC'12, June 18–22, 2012, Delft, The Netherlands.

applications, achieving higher efficiencies and lower stealing overheads.

The primary contributions of this paper are:

- A hierarchical persistence-based load balancing algorithm that performs greedy localized rebalancing
- An active-message-based retentive work stealing algorithm optimized for distributed memory machines
- First comparative evaluation of persistence-based load balancing and work stealing
- Most scalable demonstration of work stealing — on up to 163,840 cores
- Demonstration of the benefits of retentive stealing in incrementally rebalancing iterative applications

2. CHALLENGES

An effective load balancer should achieve good load balance at scale while incurring low overheads. In particular, the cost of rebalancing should be related to the degree of imbalance incurred and not the total work. In an iterative application, the load balancing overhead should decrease as the calculation evolves towards a stable state, increasing only when the application induces additional load imbalance.

Persistence-based load balancers cannot adapt to immediate load imbalance and incur periodic rebalancing overheads. If a processor runs out of work too early in a phase, it needs to wait until the end of the phase for the imbalance to be identified and corrected. Minimizing task migration for such load balancers can be beneficial by retaining data locality and topology-awareness that guided the initial distribution.

Work stealing algorithms typically employ random stealing to quickly propagate available work. This interferes with locality optimizations and topology-aware distributions. Termination detection is a challenge at scale on distributed-memory machines. While hierarchical termination detectors approximate the cost of tree-based reduce operations in principle, they incur higher costs in practice. Termination detectors run concurrent with the application, introducing additional overheads throughout the application's execution.

The cost of stealing itself is significant on a distributed-memory machine due to the associated communication latency and time to migrate the stolen work. Work stealing also ignores prior rebalancing, incurring repeated stealing costs across iterations. Due to these reasons, work stealing has traditionally been confined to shared-memory systems. In addition, given all the "noise" introduced by work stealing, it is typically employed in applications that incur significant load imbalance and are not amenable to an initial distribution. Typical approaches to distributed-memory load balancing consider hierarchical schemes due to the perceived limitations associated with work stealing.

3. RELATED WORK

Load imbalance is a well-known problem and has been widely studied in the literature. Applications involving regular data structures, such as dense matrices, achieve load balance by choosing a data distribution (e.g., multipartitioning [12]) and carefully orchestrating communication. These approaches are specialized for regular computations and do not directly extend to other classes.

Iterative calculations on less regular structures (e.g., sparse matrices [7] or meshes [28]) employ an inspector-executor approach to load balancing where the data and associated computation balance is analyzed at runtime before the start of the first iteration to rebalance the work (e.g., CHAOS [13]) Typical approaches to such start-time load balancing employ a partitioning scheme [18]. Scalable parallelization of such partitioners [8, 21] is non-trivial.

Overdecomposition is instantiated in a variety of forms by different compilers and runtimes, including Cilk [5], Intel Thread Building Blocks [24], OpenMP [16], Charm++ [20], Concurrent Collections [9], and ParalleX [17]. Many application frameworks that understand domain-specific data structures implicitly employ this approach [11].

Charm++ [20] supports a variety of persistence-based load balancing algorithms. The typical approach involves gathering statistics for objects, measuring the amount of imbalance, and executing the corresponding rebalancing algorithms in either a centralized or hierarchical [29] fashion. Such hierarchical persistence-based load balancers are employed in several scalable applications [23]. Unlike the hierarchical schemes considered by Zheng et al. [29], we focus on the development of a localized rebalancing algorithm that also incurs lower space overheads due to greedy rebalancing.

Irregular algorithms whose workload cannot be predicted at start-time, or work partitioned into sub-units, pose a significant challenge to the above load balancing approaches. Applications in this class include state-space search, combinatorial optimization, and recursive parallel codes. Work stealing is a popular approach to load balancing such applications. Cilk [5] is a widely-studied depth-first scheduling algorithm for fully-strict computations with optimal space and time bounds. It has been shown to scale well on shared memory machines and implementations are available for networks of workstations [6] and wide-area networks [27]. Prior work on extending work stealing to distributed-memory adapted the algorithm to employ remote memory access (RMA) operations using ARMCI [22], demonstrating scaling to 8192 cores [14]. An implementation in X10 extended this algorithm to reduce the interference caused by steal operations [25]. These approaches employed work stealing in a memoryless fashion. We present work stealing for distributed-memory machines using a threaded active message library developed on MPI, demonstrating scaling to significantly higher core counts.

Work stealing has not been evaluated at this scale (100,000+ cores on multiple platforms) for any application on any hardware platform using any prior algorithm. The largest prior demonstration was on up to 8192 cores [14]. The domains employing work stealing and persistence-based load balancing have traditionally been disjoint. We are not aware of any prior work comparing the effectiveness of these two schemes for iterative applications, or any other application domain.

4. PROGRAMMING MODEL

The algorithms presented in this paper are implemented in the context of a MPI-based runtime library. The runtime acts both as a user-level library for distributed memory task-based execution and as the runtime target for language constructs such as X10 `async` [10] that support non-SPMD execution modes. All processes in a group or an MPI communicator collectively switch between SPMD and task processing phases. The code snippet below illustrates the repeated execution of a task processing phase as part of an iterative application. Throughout the paper, we employ

C++ style notation except for a few shorthands in place of detailed implementation-specific API.

```
TslFuncRegTbl *frt = new TslFuncRegTbl();
TslFunc tf = frt->add(taskFunc);
TaskCollProps props;
props.functions(tf,frt)
     .taskSize(sizeof(Task))
     .localData(&procLocalObj,sizeof(procLocalObj))
     .maxTasks(localQueueSize);
UniformTaskCollSplit utc(props); //collective
for (..) {
  Task task(..); //setup task
  utc.addTask(&task, sizeof(Task)); //local operation
}
while(..) {
  utc.process(); //collective
  utc.restore(); //implicit collective
}
```

The fundamental construct in the computation is a task collection seeded with one or more tasks. The library supports several task collection variants, each specialized to exploit specific properties of the task collection known at runtime. The above example shows a task collection, called UniformTaskCollectionSplit, which optimizes for a collection of tasks of identical size, with a known upper bound on the number of tasks on any individual processor core, and additional information common across all tasks (provided as the opaque struct procLocalObj). Function pointers associated with a task execution are translated into portable handles using the function registration table, TslFuncRegTbl. These properties are used to construct a task collection object that implements the split queue work stealing algorithm, explained in Section 6. The choice of task collection can be explicitly specified by the programmer or chosen by the runtime depending on the task collection properties specified. The task collection objects are collectively created on a per-process or a per-thread basis.

The task collection objects are then seeded with tasks to begin execution. This allows for distributed locality-aware initialization. The copy overhead of task insertion in the above illustration can be avoided through in-place task initialization. The task collection, once seeded, is processed in a collective fashion using the process() method. Tasks, during their execution, can create additional tasks to be executed as part of the same or another task collection. The process() method returns when all tasks, seeded and subsequently created, have been executed and the task collection is empty, determined through a distributed termination detection algorithm. The process() method is the runtime equivalent of an X10 finish statement and corresponds to a *terminally-strict* sub-computation.

In an iterative computation, the task queue can be restored to its state prior to invocation of process() using the restore() method. This resets the termination detector and the task distribution enabling re-execution of the task collection. Execution profiles from the previous execution of the task collection can be used to adapt the seeding of tasks and scheduling algorithms employed for subsequent iterations.

5. PERSISTENCE-BASED LOAD BALANCER

On each processor core, the persistence-based load balancer collects statistical data on the durations of each task executed in the current iteration. This load database is then used to rebalance the load for the next iteration after the current iteration has terminated.

Algorithm 1 Centralized load balancer

```
peLoad ⇐ { this processor's load }
lbD ⇐ { database of tasks }
localTaskPool ⇐ { empty task pool }
sumLoad ⇐ { distributed reduction }
avgLoad ⇐ sumLoad / number of cores
while peLoad > C · avgLoad do
  task ⇐ removeSmallestTask(lbD)
  addTask(localTaskPool, task)
  peLoad -= getDuration(task)
end while
sendTo0(localTaskPool, peLoad)
if processor = 0 then
  taskPool ⇐ { received tasks }
  peLoads ⇐ { load for each PE }
  makeMaxHeap(taskPool)
  makeMinHeap(peLoads)
  while t ⇐ removeMaxTask(taskPool) do
    assign(t, getMinPE(peLoads))
  end while
  { send out new tasks to each PE }
else
  { receive new load from PE 0 }
end if
```

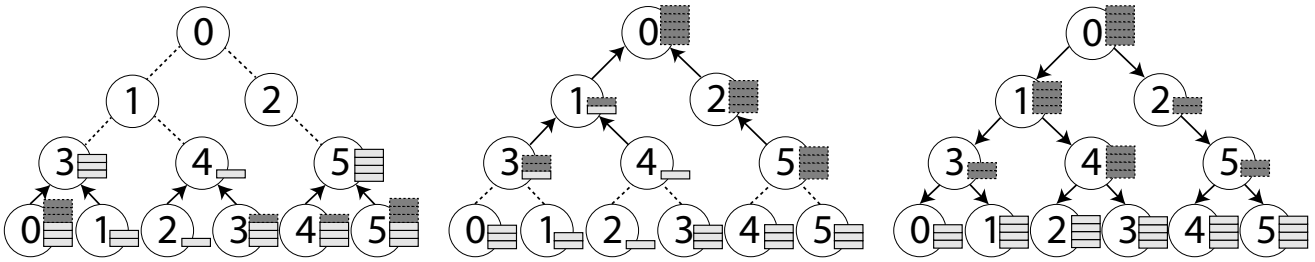
The first step in utilizing the persistence properties that an application might exhibit is collecting data. Each task that is executed by a core is timed and stored in a local database. Storing the exact duration of each task (assuming a double-precision timer), requires $\Theta(n)$ doubles and $\Theta(n)$ task descriptors, where n is the number of tasks that each core executes. To reduce the amount of storage required, the scheduler times each task, truncates the duration, and bins it with other tasks that are of approximately the same duration. This reduces the storage to $\Theta(n) + b$, where n is the number of task descriptors and b is the number of bins. These bins are kept in a sorted structure, allowing the load balancer to access the database in roughly duration order.

5.1 Centralized Load Balancing

The baseline strategy, referred to as the centralized scheme, performs load balancing on one core, much like RefineLB as described by Zheng et al. [29]. Algorithm 1 details this strategy. First, the average load is calculated in parallel using a sum-reduction. If a core's load is sufficiently above the average load, the core removes the shortest duration task from its load database and moves it into a local task pool, until its load is below a constant, referred to as C, times the average load. These tasks (descriptors that compactly represent the task) in the local task pool, along with the core's new load, are sent to core 0, which attempts to redistribute the load.

Core 0 receives the donated tasks from overloaded cores, storing them into a task pool. It also receives each core's total load. From this data it creates a min-heap of the loads of each core and a max-heap of the durations in the task pool. It then assigns the longest task to the most underloaded core until the task pool is empty. These assignments are then sent to the cores.

To maximize the scalability of this approach, the local task pool from each core is collected on core 0 using MPI_Gatherv, and new assignments are redistributed using MPI_Scatterv.



(a) All the leaves of the tree send their excess load (any load above a constant, referred to as C, times the average load) to their parent, selecting the work units from shortest duration to longest.

(b) Each core receives excess load from its children and saves work for underloaded children until their load is above a constant D times the average load. Remaining tasks are sent to the parent.

(c) Starting at the root, the excess load is distributed to each child applying the centralized greedy algorithm: maximum-duration task is assigned to minimum-loaded child.

Figure 1: The hierarchical persistence-based load balancing algorithm for 6 cores. The rectangles represent work units; a shaded rectangle with a dotted border indicates the work unit moves during that step.

5.2 Hierarchical Load Balancing

Algorithm 2 details the procedure and Figure 1 illustrates the structure of the load-balancing tree. The cores are organized as an n-ary tree where every core is a leaf. First, the average load across all cores is determined. Each core locally chooses its shortest tasks to donate while reducing its anticipated load to be below the average times a threshold.

An underloaded core contains an empty set of tasks to be donated. Each core then sends its load information together with the set of donated tasks to its parent. Each core above the first level in the tree then receives the tasks and load from each child. These cores redistribute the donated tasks to balance the lightly loaded cores using the same procedure as the centralized algorithm. Each core then sends the total surplus or deficit in its sub-tree, together with donated tasks left over from the distribution, to its parent. This algorithm is repeated recursively up the tree to the root. The root redistributes leftover tasks down the tree to further improve load balance. Any tasks provided by a core's parent are redistributed down the tree to the leaf nodes, where they are enqueued for the next iteration.

This algorithm is greedy since it locally optimizes for the children of a node as work moves up the tree, assigning tasks to underloaded children, and down the tree to distribute the work. The average child load (total load of that subtree divided by the subtree size) at each node is used to make a local decision about which child to assign work. Such a greedy approach reduces communication and the amount of storage required at each level by considering only the immediate children and assigning them work immediately. This also reduces the amount of time to rebalance at each level because fewer tasks must be considered. In addition, tasks are assigned to locally deficit cores, better preserving data and topology locality than the centralized rebalancing scheme.

On the other hand, the quality of the load balance may diminish when the tree is extremely unbalanced because local decisions based on averages do not cause work to be migrated aggressively. Also, when there is a great disparity in the work unit size, large work units may be assigned to a locally deficit core rather than the most underloaded core, not effectively addressing the load imbalance problem.

Note that tasks themselves can be distributed directly from the donating core to the designated core, rather than through the tree. The greedy algorithm can also be applied to other organizations of the cores (e.g., torus) to better match the underlying communication network's topology.

Algorithm 2 Hierarchical greedy load balancer

```
peLoad ⇐ { this processor's load }
lbD ⇐ { database of tasks }
localTaskPool ⇐ { empty task pool }
sumLoad ⇐ { distributed reduction }
avgLoad ⇐ sumLoad / number of cores
while peLoad > C · avgLoad do
  task ⇐ removeSmallestTask(lbD)
  addTask(localTaskPool, task)
  peLoad -= getDuration(task)
end while
sendToParent(localTaskPool, peLoad)
{ wait for children }
if processor is not root then
  { wait for children }
  taskPool ⇐ { received tasks }
  peLoads ⇐ { load for each child PE }
  makeMaxHeap(taskPool)
  makeMinHeap(peLoads)
  while t ⇐ removeMaxTask(taskPool) ∧
        getMinPELoad(peLoads) < D · avgLoad do
    assign(t, getMinPE(peLoads))
  end while
else
  { wait for children }
end if
if processor is not leaf then
  taskPool ⇐ { received tasks }
  peLoads ⇐ { load for each child PE }
  makeMaxHeap(taskPool)
  makeMinHeap(peLoads)
  while t ⇐ removeMaxTask(taskPool) do
    assign(t, getMinPE(peLoads))
  end while
  for all p in children do
    sendTasks(p)
  end for
else
  { wait to receive tasks }
  { add tasks to pool }
end if
```

6. RETENTIVE WORK STEALING

In this section, we present our implementation of distributed memory work stealing. Work stealing begins with all participating cores seeded with zero or more tasks. A core with local work takes the role of a worker, processing local tasks, as long as they are available. Once local work is exhausted, a worker becomes a thief, searching for other available work. A thief randomly chooses a victim and attempts to steal work from the victim's collection of tasks. On a successful steal, a thief returns to the worker state, continuing to process its local tasks. This procedure repeats until all workers have exhausted their tasks and termination is detected.

The randomness in the choice of the victim ensures quick distribution of work even in highly imbalanced cases (e.g., only one worker starts with all the work). If sufficient parallel slack is present, generally more cores will be in worker state rather than searching for work as a thief. Therefore, work stealing implementations, as pioneered by Cilk, attempt to avoid the overheads of synchronization or atomic operations on the executing worker, forcing much of the synchronization overhead on the thieves.

Shared memory implementations employ a deque in which the worker inserts the tasks on one end (referred to as the *head*), while thieves steal from the other end (the *tail*). Fence instructions are employed by the worker to ensure that its insertions at the head are visible to potential thieves in the correct order. A thief obtains a lock on the deque to preclude other concurrent steals. More details of the algorithm can be found in Blumofe et al. [5]

We employ a distributed-memory work stealing algorithm that considers the differing costs involved in distributed-memory machines.

```
Task taskBuf[BSIZE];//array holding tasks on the deque
Lock lock; //lock to arbitrate access to the deque
int head; //head: accessed only by worker
volatile int split; //split: worker reads/writes; thief reads
volatile int stail; //position of next steal; thief reads/writes;
      worker reads
volatile int itail; //intermediate tail: worker reads, thief writes
int ctail; //completed tail: accessed only by worker
Initial values:
  lock.unlock();
  head = split = stail = itail = ctail = 0;
```

A distributed-memory implementation of work stealing requires tasks to be copied to local memory, rather than just obtaining a pointer. On many architectures, such data transfer is more efficient from a contiguous memory rather than an arbitrary data structure (such as a linked list of tasks). We therefore employ a bounded-buffer circular deque implementation on a fixed-sized array.

The operations on the deque are depicted in Figures 2, 3, and 4. The dotted arrows correspond to locked or atomic operations, while the vertical dotted lines depict updated values for the variables being modified.

Each worker (a processor core in our case) allocates such a deque and associated state variables. The reuse of taskBuf allows it to reside in "registered" memory, enabling efficient data transfer. Rather than allowing all tasks in the deque to be stolen by a thief, we employ a split deque. All tasks between head and split are local to the worker owning the deque and cannot be stolen by a thief. This enables a worker to add and remove tasks at the head, without the potential need for fence instructions. The tasks past the split are in the shared portion of the deque and are available for stealing by thieves.

Individual remote memory operations (obtaining a lock, adjusting indices, releasing a lock, etc.) incur significant latencies. This not only increases the cost of a steal but slows down work propagation by precluding other steals. In order to enable contesting thieves to make quick progress, a split-phase stealing protocol is employed so that stolen tasks can be concurrently transferred while other thieves make progress. The shared portion between split and stail represents tasks that are available to be stolen. The shared portion between stail and ctail corresponds to stolen tasks that are being copied into the thieves' local memories.

The state of the deque can be identified by the following:

```
bool full() {
  return head==ctail && (split!=head || split!=ctail);
}
bool sharedEmpty() { return split==stail; }
bool localEmpty() { return head==split; }
int sharedSize() { return (split−stail+BSIZE)%BSIZE;}
int localSize() { return (head−split+BSIZE)%BSIZE;}
```

Note that the state of the deque queried without holding a lock is speculative if any of the variables associated with computing the state is marked as volatile. The only exception is split, which can only be modified by the worker and therefore can be read by it without holding the lock.

Adding a task into the deque by a worker (addTask()) involves inserting the task at the head. The method also resets the space available for insertion by adjusting ctail. This employs the invariant:

$$(\text{itail}==\text{stail}) \equiv \text{no pending steals on taskBuf[stail..ctail]} \quad (1)$$

```
void releaseToShared(int sz) {
  memfence();
  split = (split+sz)%BSIZE;
}
void addTask(Task task) {
  do {
    if(itail==stail) ctail = stail;
  } while(full());
  taskBuf[head] = task;
  head = (head+1)%BSIZE;
  if (sharedEmpty())
    releaseToShared(localSize() / 2);
}
```

When a worker observes the shared portion of its deque to be empty, it releases tasks from its local portion, shown by the routine releaseToShared(). Note that a memory fence operation is required while adding or getting tasks only when releasing work to the shared portion of the deque.

```
bool acquireFromShared() {
  lock();
  if(sharedEmpty()) return false;
  int nacquire = min(sharedSize()/2,1);
  split = (split − nacquire + BSIZE) % BSIZE;
  unlock();
  return true;
}
bool getTask(Task *task) {
  if(localEmpty())
    if(!acquireFromShared()) return false;
  *task = taskBuf[head];
  head = (head − 1 + BSIZE) % BSIZE;
  if (sharedEmpty())
    releaseToShared(localSize() / 2);
  return true;
}
```

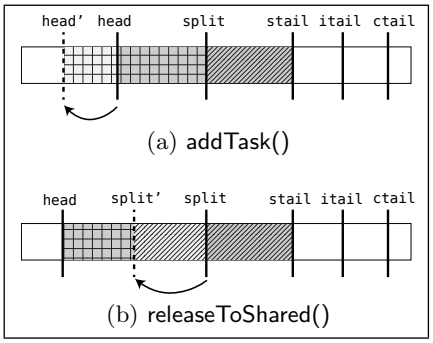


Figure 2: Routines related to adding tasks into the deque

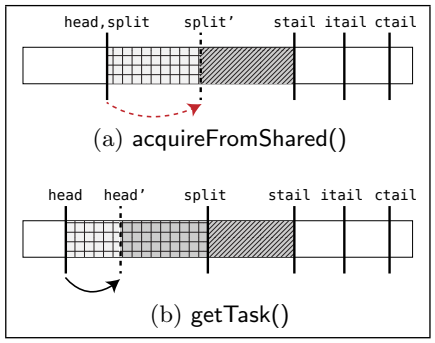


Figure 3: Routines for a worker getting tasks from its deque

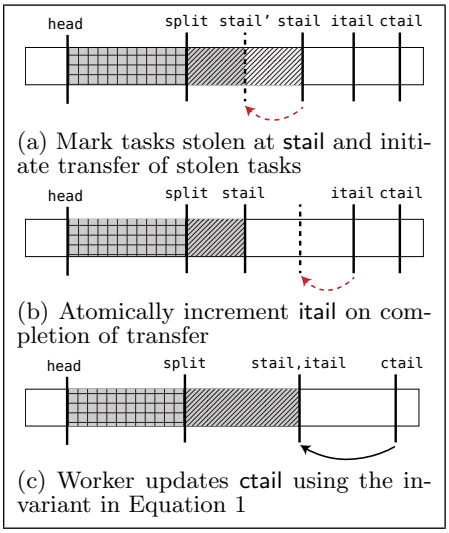


Figure 4: Steps in a steal operation

	HF-Be256	HF-Be512 (20/40)	TCE
Total tasks	84.9M	21.7B / 1.36B	470K
Non-null tasks	213K	9.10M / 862K	470K

Table 1: Number of tasks in first (all tasks) and subsequent iterations (non-null tasks). The chunk size for HF-Be512 is shown in parentheses.

Extracting a task from the deque first involves transferring tasks from the shared to the local portion, if the local portion is empty. The task at the head is then returned.

When a worker runs out of local work (getTask() returns false), it attempts to steal work from a random worker proc. This is implemented as an active message executed on the victim. The block of code executed as an active message is annotated by @proc in the code snippet. The victim's deque is locked and any tasks to be stolen are marked (using stail) before unlocking the deque. The stolen tasks are then transferred to the thief through an asynchronous operation (sendResponse()). When that operation is complete (detected by other runtime system components), the completion is noted in the deque by atomically updating itail. Note that multiple thieves could initiate the transfer of stolen tasks and complete the transfers out-of-order. Hence, ctail, which denotes completed steals, cannot be updated by a thief upon completion of its transfer, but needs to be updated by the worker using the invariant shown in Equation 1.

```
bool steal(int proc) {
  if(hasTerminated()) return false;
  //post a recv for incoming response
  @proc { //active message executed on proc
    lock();
    if(sharedEmpty()) { sendResponse(NULL); }
    int nsteal = min((sharedSize()+1)/2, BSIZE−stail);
    int oldtail = stail;
    int newtail = (stail + nsteal) % BSIZE;
    int nshift = newtail − oldtail;
    stail = newtail;
    unlock();
    sreq = sendResponse(taskBuf[oldtail..(oldtail+nsteal)]);
    when sreq.localComplete() {
      atomic itail += nshift;
    }
  }
  //wait on recv
  if(recvSize()>0) {
    lock();
    // adjust head, split, ...
    unlock();
    return true;
  }
  return false;
}
```

The stolen tasks are directly transferred from the victim's deque to the thief's deque, without additional copy operations. In the meanwhile, any attempts to steal from the thief would fail since the state variables denote the deque to be empty. On completion of a successful data transfer, and receipt of non-zero number of tasks, the state variables are updated under a lock to denote the availability of tasks.

```
void process() {
  while (!hasTerminated()) {
    Task task;
    while (getTask(&task)) {
      executeTask(&task);
    }
    bool got_work = false;
    while(got_work == false && !hasTerminated()) {
      do {
        p = rand() % nproc();
      } while (p == me());
      got_work = steal(p);
    }
  }
}
```

The overall execution procedure is shown above. All participating processor cores execute this routine. Each core executes all local tasks in the worker role, then turns into a thief searching for work. On finding work, the thief turns back into a worker. This cycle is repeated until termination is detected.

The tasks executed by each core are logged throughout the execution. In the next iteration, the processing begins with each core's local queue seeded with tasks executed by it in the previous iteration. Given that work stealing attempts to dynamically balance the load each time it runs, this retained task distribution is anticipated to be more balanced than the initial seeding.

Essentially by retaining the previous execution profile, the advantages of persistence-based load balancing are applied to work stealing, and depending on the degree to which the persistence principle applies, the number of attempted and successful steals diminish in subsequent iterations with this optimization (shown in Figures 8 and 9). This strategy also maintains the primary advantage of work stealing: the abil-

ity of the runtime to adapt to major dynamic imbalances during an iteration.

```
set<Task> executedTasks;
void executeTask(Task task) {
  executedTasks.insert(task);
  // execute task
}
void restore() {
  foreach task in executedTasks {
    addTask(task);
  }
  executedTasks.clear();
}
while(..) {
  process();
  restore();
}
```

Note that the balance determined by the work stealing algorithm includes associated overheads. Therefore significant imbalance, together with work stealing, can still be expected in subsequent iterations. In Section 7, we show that this approach incrementally improves the load balance while also reducing the work stealing overheads.

Termination detection is done using Francez's termination detection algorithm [15], involving a tree in which thieves propagate termination messages in the form of rounds up and down the tree. Any round is invalidated by a thief that was a victim of a steal since the last round. Termination is detected when all workers have turned thieves, participated in the termination detection procedure, and none of them have been stolen from since the last round. The organization of the workers/thieves into a tree results in a theoretical logarithmic overhead of termination detection after all tasks have been executed.

7. EXPERIMENTAL EVALUATION

The experiments were performed on three systems: Cray XE6 NERSC Hopper [2], IBM BG/P Intrepid [1], and Cray XK6 Titan [3]. Hopper is a 6384-node system, each node consisting of two twelve-core 2.1GHz AMD 'MagnyCours' processors and 32GB DDR3 memory. Titan is a 18688-node system, each node consisting of one sixteen-core 2.2GHz AMD 'Bulldozer' processor and 32GB DDR3 memory. Hopper and Titan employ the Gemini interconnection network with a peak of 9.8GB/s bandwidth per Gemini chip. The Intrepid system consists of 40960 nodes, each with one quad-core 850MHz PowerPC 450 processor and 2GB DDR2 memory.

Our codes were compiled using the Intel compiler suite versions 12.0.4.191 and 12.1.1.256 on Hopper and Titan, respectively. Cray MPICH2 XT versions 5.3.3 and 5.4.1 were used on Hopper and Titan, respectively. On Intrepid, our codes were compiled with IBM XL C/C++ version 9.0. All communication was performed using two-sided MPI operations (MPI_Isend(), MPI_Irecv(), and MPI_Wait()), except the collectives employed in the persistence-based load balancing, as specified in Section 5. We developed a thread-based implementation with one thread pinned to each core throughout the execution, all of them sharing data, with MPI initialized in MPI_THREAD_MULTIPLE mode. We evaluated various configurations by varying the number of worker threads and "server" threads, and found that the best performance (in all the configurations we evaluated) was achieved with 23 worker threads and 1 server thread on Hopper, 15 worker threads and 1 server thread on Titan, and 3 worker threads and 1 server thread on Intrepid. We report all our results for these configurations. Given the server thread is still employed in communication progress for the application, we report all results as if the application is utilizing all the cores.

We evaluated the following schemes:

- **StealAll:** Work stealing of all tasks in the calculation
- **Steal:** Work stealing non-null tasks (same as StealAll for TCE)
- **StealRet:** Retentive work stealing on non-null tasks
- **PLB:** Persistence-based load balancing
- **Ideal:** Ideal scaling expected, for comparison

7.1 Benchmarks

The algorithms presented were evaluated using the following two benchmarks:

Tensor Contraction Expressions: Tensor Contraction Expressions (TCE) [4] comprise the entirety of Coupled Cluster methods, employed in accurate descriptions of many-body systems in diverse domains. Tensors are generalized multi-dimensional matrices organized into dense rectangular tiles. A tensor contraction can be viewed as a collection of tile-tile products. The sparsity in the tensors, which can be algebraically determined through inexpensive local integer operations, induces inhomonegenity in the computation of dense tile-tile contractions, which vary widely in their computational requirements, spanning in structure from inner products to outer products.

Hartree-Fock: The Hartree-Fock (HF) method is a single-determinant theory [26] that forms the basis for higher-level electronic structure theories such as Møller-Plesset perturbation theory and Coupled Cluster theory. The benchmark consists of the two-electron contribution component of Hartree-Fock, the computationally dominant part of the method. The work to be performed is divided into smaller units based on the user-specific tile size. The work to be performed is determined by the schwarz screening matrix. Unlike the TCE benchmark, the sparsity induced by the schwarz matrix depends on the specific input and cannot be determined at compile-time. The screening produces variability in the execution time of individual tasks and potentially results in *null* tasks, i.e., tasks that do not perform any work. These null tasks are pruned in subsequent iterations.

For the Hartree-Fock benchmark, we considered two different molecular systems for evaluation, one consisting of 256 beryllium atoms (HF-Be256) and the other 512 atoms (HF-Be512). The matrices involved in the Hartree-Fock calculation (schwarz, fock, and dens matrices) were block-cyclically distributed amongst the cores with a tile size of 40 for evaluation on Hopper. A tile size of 20 was used on Titan and Intrepid to expose additional parallelism and enable evaluation on larger core counts. The number of total and non-null tasks is shown in Table 1. Note that the number of non-null tasks quadruples when the number of atoms is doubled. The tasks themselves do not necessarily incur the same execution time due to the sparsity induced by the schwarz matrix.

For TCE, we evaluate the following expression:

$$C[i,j,k,l] += A[i,j,a,b] * B[a,b,k,l]$$

Each dimension is split into four spatial blocks. Indices i, j, k, and l are organized into spatial blocks $240, 180, 100$, and 210; indices a and b are of size 84 and are organized into spatial blocks $20, 24, 20$, and 20. The tensors are partitioned into dense blocks with a tile size 20 and distributed in the

global address space. Detailed explanation of the benchmark can be found in Baumgartner et al. [4]

We observed that applications converged faster to the best achievable efficiency on Hopper and present five iterations for each application. Both schemes required many more iterations to converge to the best possible efficiency on Titan and Intrepid. We present results for the first fourteen iterations on these systems. Complete results for all configurations considered are not presented due to space limitations.

Persistence-based load balancing is typically performed only when significant load imbalance is detected to amortize the cost of load balancing. We load balance every iteration to quickly evaluate the effectiveness of load balancing. For the experiments, the load balancer constants C (Algorithm 1) and D (Algorithm 2) were set to 1.003; the branching factor used in the hierarchical version was 3. The results presented do not include the load balancing cost, which is evaluated separately in section 7.3.

7.2 Scalability and Efficiency

The execution times of the various schemes for both applications are shown in Figure 5. For the HF benchmark, the execution time for stealing all tasks corresponds to the zero-th iteration of all runs, before pruning the null tasks, while other times presented correspond to the last iteration for the respective runs. Lines marking ideal speedup (with respect to the smallest core count shown in the corresponding graph as the baseline) are shown for comparison. For the HF benchmark, executing all tasks is significantly more expensive than executing only the non-null tasks. This is primarily due to the communication and computation associated with identifying the null tasks (checking the schwarz matrix). The schemes also scale well with increase in core count. Traditional work stealing scales better when executing all tasks, given the increased degree of available parallelism. This demonstrates that random work stealing can scale to large core counts provided sufficient parallelism.

When executing only the non-null tasks, work stealing is less scalable than the other approaches evaluated. Persistence-based load balancing and retentive stealing achieve the best performance. The TCE benchmark exhibits super-linear scaling due to the working set fitting in the cache. This effect is countered by load imbalance at higher scales. Retentive work stealing appears to address the problems associated with work stealing.

In order to better evaluate the observed performance and evolution of the schemes with iterations, we plotted their efficiency for each of the iterations on the non-null tasks. For each problem size, the efficiency is measured with respect to the best performance achieved by any of the schemes at any core count considered for that problem size.

The efficiency of retentive work stealing is shown in Figure 7. The efficiency of traditional work stealing is that achieved by the first iteration. Retentive work stealing steadily improves its efficiency with subsequent iterations; for HF-Be512 the best points achieve over 90% efficiency on 76,800 cores of Hopper, over 91% efficiency on 163,940 cores of Intrepid, and over 81% efficiency on 128,000 cores of Titan. Given that the effectiveness of work stealing is influenced by the parallel slack, we plot the average number of tasks, with error bars showing the standard deviation across different processor cores, for each core count. This plot shows the similarity of task distributions across the problem sizes. For each problem size, the efficiency begins to deteriorate when less than 10 tasks per processor core are available on average. For a given number of tasks per processor core, the efficiency decreases with core count.

7.3 Cost and Effectiveness of Persistence-Based Load Balancers

In this section, we evaluate the effectiveness of the centralized and hierarchical persistence-based load balancers using micro-benchmarks constructed from the execution times for each task in the HF-Be256 run on Hopper. In order to simulate various degrees of load imbalance and stress the load balancers, the tasks are sorted by duration and distributed to the cores in a round-robin fashion, favoring every n-th core by giving it m tasks. For example, when $n = 2$ and $m = 4$ every other core receives 4 tasks, instead of 1 task.

By varying m and n, the performance of both schemes is evaluated under severe amounts of load imbalance. As the problem is scaled on more cores, due to the number of tasks remaining constant, much more load imbalance arises because there is more variance in duration between the tasks that each core selects. For example, when $n = 4$ and $m = 4$, on 2400 cores, the core with the highest load has 252% the average amount of work; on 38400 cores, the highest loaded core has 691% the average. As illustrated by these data points, the imbalance increases with the number of cores, necessitating that substantially more tasks be migrated.

Table 3 shows execution times and quality of the load balance achieved by both schemes on Hopper. The quality of load balance is measured as percentage deterioration in the execution time over the ideal load balance. The execution time for a given distribution of tasks is defined by the load on the core with the maximum load, while the ideal is the average of the load across all cores. Since the ideal load balance is not always achievable, we compare the hierarchical algorithm with the centralized algorithm.

The centralized load balancer achieves a consistently good load balance, given its global view of the computation at all scales. The variations in the load balance quality are due to the differences in tasks assigned to different cores. The hierarchical algorithm suffers from worse load balance quality due to the greedy rebalancing employed. At scale, the few tasks available per core exacerbates the challenges encountered by greedy rebalancing. However, it is still competitive given sufficient number of tasks to rebalance. The execution times of the load balancers clearly demonstrate the benefits of the hierarchical approach.

7.4 Quantifying Work Stealing Overheads

In order to study the improvements in performance obtained by retentive stealing, we measured the number of attempted and successful steals for the various problems. Figure 8 and Table 2 show the number of attempted steal operations for the first, second, and fifth iterations. We observe that the number of steals, of any form, does not increase dramatically with an increased process count. For example, on iteration 1 of HF-Be512 on Hopper, the average number of attempted steals decreases from 5661 on 9600 cores to 5471 on 38400 cores.

If sufficient parallelism is present, the number of steals is influenced more by the problem size and initial task distribution than the number of concurrent executing cores. When the degree of application parallelism dries out, as happens in

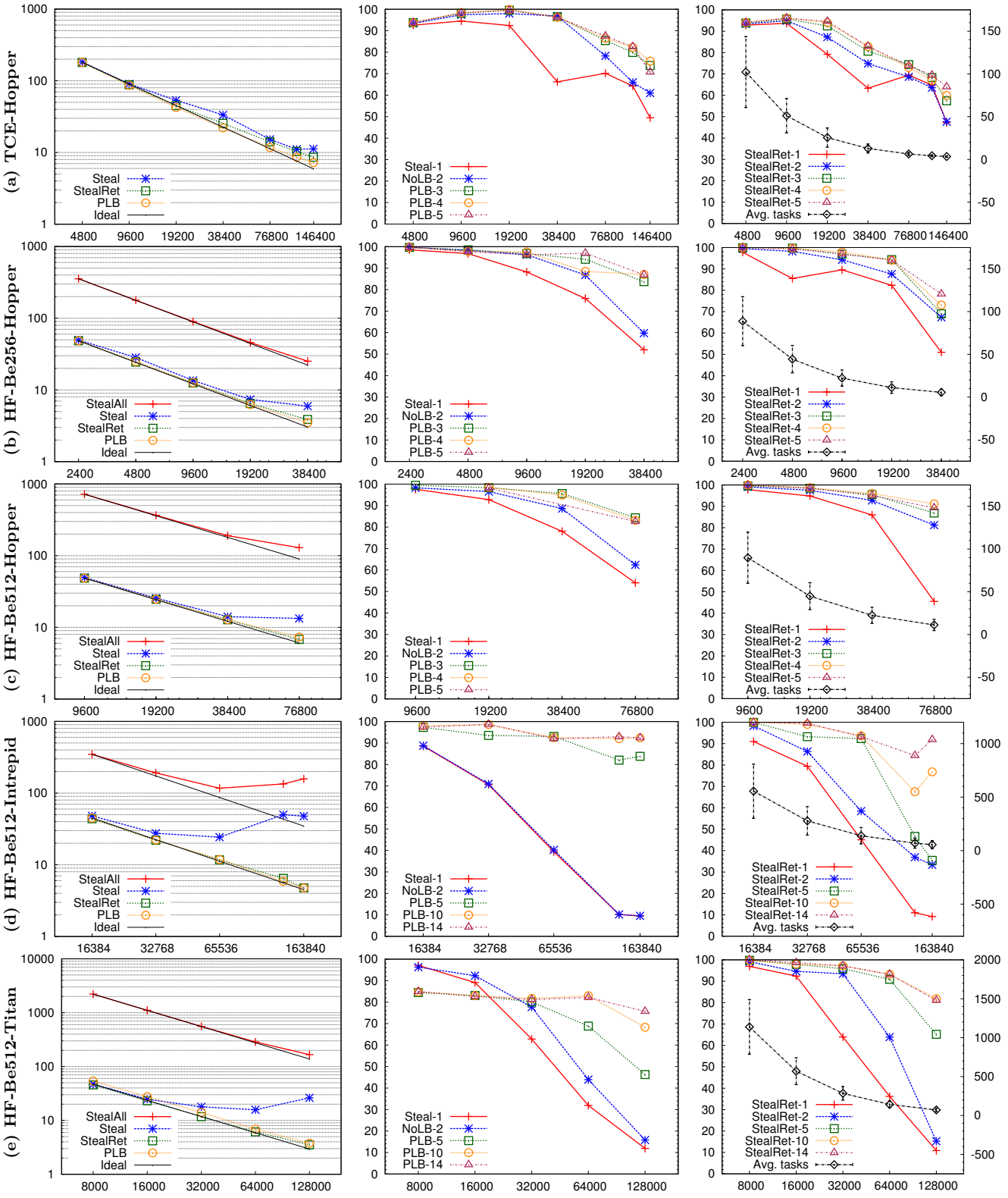


Figure 5: Execution times for first and last iteration. x-axis — number of cores; y-axis — execution time in seconds

Figure 6: Efficiency of persistence-based load-balancing across iterations for the three system sizes, relative to the ideal anticipated speedup. x-axis — number of cores; y-axis — efficiency

Figure 7: Efficiency of retentive work stealing across iterations relative to ideal anticipated speedup and tasks per core. x-axis — core count; left y-axis — efficiency; right y-axis — tasks per core (error bar: std. dev.)

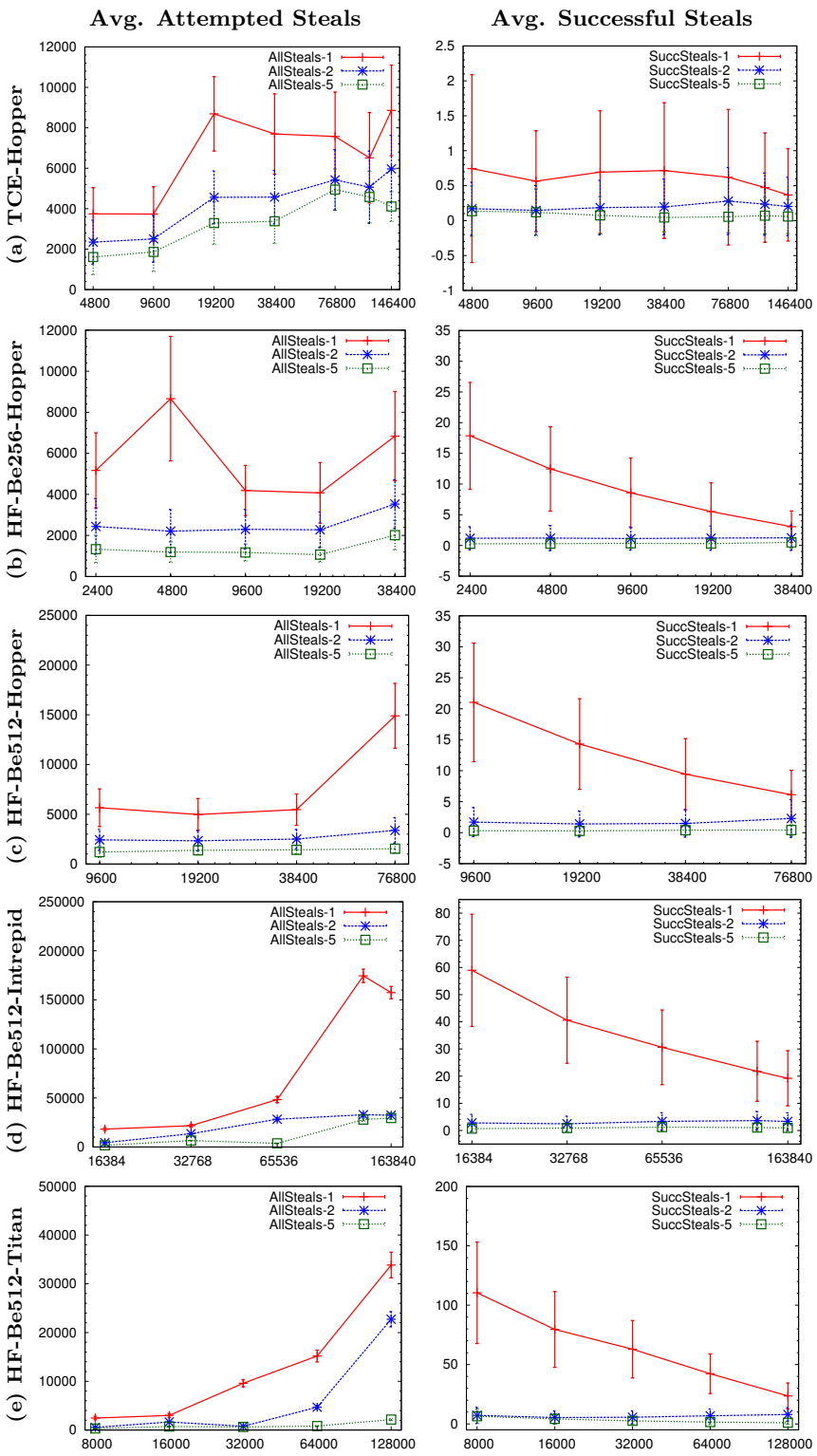


Figure 8: Average (error bar: standard deviation) number of attempted steals for the first, second, and fifth iteration of retentive stealing. x-axis — number of cores; y-axis — average number of steals.

Figure 9: Average (error bar: standard deviation) number of successful steals for the first, second, and fifth iteration of retentive stealing. x-axis — number of cores; y-axis — average number of steals.

Iter	Avg / Std Attempted	Avg / Std Successful	Avg / Std Attempted	Avg / Std Successful
	4800 Cores		9600 Cores	
1	3745/1294	0.7/1.3	3735/1350	0.6/0.7
2	2347/1086	0.2/0.4	2507/1155	0.1/0.4
5	1610/868	0.1/0.4	1864/964	0.1/0.3
	19200 Cores		38400 Cores	
1	8691/1840	0.7/0.9	7694/1987	0.7/1.0
2	4563/1306	0.2/0.4	4570/1325	0.2/0.4
5	3291/1049	0.1/0.3	3381/1095	0.04/0.2
	78600 Cores		146000 Cores	
1	7573/2196	0.6/1.0	8859/2238	0.4/0.7
2	5426/1496	0.3/0.5	5967/1663	0.2/0.4
5	4944/1019	0.1/0.2	4109/739	0.1/0.2

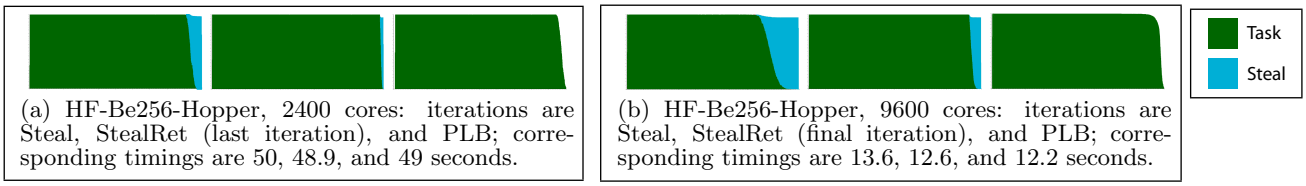
	2400 Cores		4800 Cores	
1	5168/1831	18/8.7	8662/3034	12/6.9
2	2436/1356	1.2/1.9	2209/1056	1.2/2.1
5	1333/667	0.3/0.7	1192/498	0.3/0.7
	9600 Cores		19200 Cores	
1	4190/1221	8.6/5.7	4073/1476	5.5/4.7
2	2298/966	1.1/1.9	2275/873	1.2/2
5	1171/412	0.3/0.8	1066/360	0.3/0.7
	38400 Cores			
1	6837 / 2178	3.1 / 2.6		
2	3528 / 1186	1.2 / 2.1		
5	2015 / 715	0.5 / 0.9		

	9600 Cores		19200 Cores	
1	5661/1897	21/9.6	4975/1602	14/7.3
2	2428/1047	1.7/2.4	2336/1020	1.4/2.1
5	1202/549	0.3/0.8	1375/457	0.3/0.7
	38400 Cores		76800 Cores	
1	5471/1571	9.4/5.7	14899/3267	6.1/4
2	2512/965	1.5/2.2	3382/1278	2.3/3.1
5	1430/412	0.4/0.8	1537/439	0.4/0.9

	16384 Cores		32768 Cores	
1	18220/1569	59.0/20.6	21736/1615	40/15
2	4386/633	2.8/3.1	13522/844	2.5/2.8
5	1697/231	0.7/1.1	6548/373	0.9/1.3
	65536 Cores		131072 Cores	
1	48315/3026	30/14	174496/6860	21/11
2	28249/1583	3.4/3.2	32904/1793	3.7/3.4
5	3694/318	1.2/1.6	27914/1620	1.1/1.5
	163840 Cores			
1	157353/6307	19/10		
2	32368/1813	3.3/3.2		
5	29479/1516	1/1.4		

	8000 Cores		16000 Cores	
1	2461/356	110/42	2994/410	79/31
2	538/113	7.3/6.9	1652/142	5.4/5.5
5	320/87	6.5/6.1	714/84	4.2/4.5
	32000 Cores		64000 Cores	
1	9592/713	63/24	15180/1225	42/17
2	770/118	5.6/5.3	4680/412	6.8/5.7
5	631/76	2.6/3.1	814/86	1.6/2.1
	128000 Cores			
1	33844/2634	24/11		
2	22725/1571	8.0/5.8		
5	2155/217	1.0/1.4		

Table 2: Average (Avg) number and standard deviation (Std) of attempted and successful steals for retentive work stealing for TCE, HF-Be256, and HF-Be512 benchmarks on Hopper Cray XE6, Intrepid IBM BG/P, and Titan Cray XK6.



(a) HF-Be256-Hopper, 2400 cores: iterations are Steal, StealRet (last iteration), and PLB; corresponding timings are 50, 48.9, and 49 seconds.

(b) HF-Be256-Hopper, 9600 cores: iterations are Steal, StealRet (final iteration), and PLB; corresponding timings are 13.6, 12.6, and 12.2 seconds.

Figure 10: Utilization for all worker threads over time. x-axis — time; y-axis — percent utilization.

	Load Balancer Execution Time (seconds)					Load Balancer Quality (percent over ideal)				
n m	2400 C / H	4800 C / H	9600 C / H	19200 C / H	38400 C / H	2400 C / H	4800 C / H	9600 C / H	19200 C / H	38400 C / H
1 1	1.57 / 1.42	1.98 / 1.12	5.76 / 1.08	14.8 / 1.92	35.6 / 1.04	0.03 / 0.06	0.03 / 0.18	0.03 / 0.4	0.03 / 2.0	5.8 / 12
2 2	4.83 / 0.07	6.33 / 0.16	8.42 / 0.81	13.0 / 0.77	43.8 / 1.00	0.03 / 0.14	0.03 / 0.54	0.03 / 1.1	1.7 / 6.6	5.8 / 12
2 4	7.40 / 0.35	10.0 / 0.66	13.0 / 0.33	18.3 / 0.31	37.7 / 1.14	0.03 / 0.26	0.03 / 0.68	0.03 / 4.0	0.03 / 7.1	5.8 / 12
2 8	9.24 / 0.34	11.4 / 0.30	13.0 / 0.33	18.4 / 1.62	45.2 / 4.40	0.03 / 0.28	0.03 / 2.2	0.03 / 3.7	0.1 / 7.3	5.8 / 18
4 2	1.40 / 0.08	2.00 / 0.30	3.82 / 0.48	16.9 / 0.82	39.5 / 0.75	0.03 / 0.17	0.03 / 0.53	0.03 / 1.3	2.8 / 5.7	5.8 / 11
4 4	2.84 / 0.09	4.13 / 0.20	6.65 / 0.56	13.4 / 0.36	37.0 / 0.65	0.03 / 0.29	0.03 / 0.60	0.03 / 2.4	1.3 / 7.4	5.8 / 15
4 8	4.34 / 0.28	6.27 / 0.97	9.07 / 0.58	15.1 / 0.26	43.0 / 0.85	0.03 / 0.29	0.03 / 0.58	0.03 / 3.9	0.03 / 6.9	5.8 / 13
8 2	0.45 / 0.04	0.75 / 0.11	2.73 / 0.25	15.0 / 0.79	36.2 / 0.36	0.03 / 0.30	0.03 / 0.73	0.03 / 1.2	2.9 / 5.4	6.6 / 13
8 4	0.98 / 0.06	1.52 / 0.07	3.03 / 0.19	11.9 / 0.65	38.1 / 0.55	0.03 / 0.50	0.03 / 1.0	0.03 / 3.1	2.3 / 7.6	5.8 / 12
8 8	1.88 / 0.06	2.70 / 0.10	5.01 / 0.49	11.4 / 0.38	36.7 / 0.63	0.03 / 0.49	0.03 / 1.0	0.03 / 4.6	0.9 / 7.9	5.8 / 15

Table 3: LHS — execution time (seconds) on Hopper for rebalancing 12 initial distributions of tasks of the HF-Be256 system, comparing centralized (C) and hierarchical (H) persistence-based schemes. RHS — load balance quality, computed as the maximum percentage over the ideal execution time (perfectly balanced load). The ideal execution times are 48, 24, 12, 6, and 3 seconds, respectively.

our application in the highest scale considered for each problem, steal attempts significantly increase. For iteration 1 of HF-Be512 on Hopper, increasing the number of cores from 38400 to 76800 increases the average number of attempted steals from 5471 to 14898. This trend is due to a lack of parallel slack, not just the increasing core counts. The high standard deviation shows that different cores are provided with different initial loads and attempt a varying number of steals to find work.

The number of attempted steals provides insight into the improvements in the performance achieved by retentive stealing. As the load balancing becomes iteratively refined with retentive stealing, the number of attempted steals decreases and stabilizes across process counts. In addition, the gradual balancing of the load is accompanied by a lower standard deviation. Intuitively, when each processor core finishes the work initially assigned to it, all cores reach a similar state and the entire application is close to completion. For iteration 5 of HF-Be512 on Hopper, increasing the number of cores from 38400 to 76800 after retentive stealing has been applied for several iterations, only increases the average number of attempted steals from 1430 to 1537.

The number of successful steals, shown in Figure 9 and Table 2, confirms our intuition from the number of attempted steals. A successful steal depends on the availability of pending work, which decreases with increase in core count. More importantly, retentive stealing balances the load well enough that for the last iteration the number of successful steals is very small. For iteration 5 of HF-Be512 on Hopper, for core counts 9600, 19200, 38400, and 76800, the average number of successful steals are 0.3, 0.3, 0.4, and 0.4, respectively.

The efficiency of persistence-based load balancing, using the hierarchical load balancer, is shown in Figure 6. The first iteration plotted on the graph is the efficiency of work stealing after the null tasks have been pruned. The significant change in the work distribution from iteration 0 to 1 renders persistence-based load balancing ineffective, resulting in large observed execution times. Instead, we resorted to work stealing in the first iteration, in this evaluation.

For the second iteration, to show the overhead of work stealing, tasks are seeded based on the previous iteration, but work stealing is disabled and no load balancer is used. Efficiency improves slightly in this case because the overheads associated with stealing and termination detection are removed. In subsequent iterations, the hierarchical load balancer is executed before the start of the iteration, using the data collected from the previous iteration. The first time the hierarchical load balancer is run, load balance improves significantly; for example, on Hopper efficiency increases by 7% on 76800 cores for the TCE calculation, 24% on 38400 cores for the HF-Be256 system, and 22% on 76800 cores for the HF-Be512 system. The second application of the load balancer improves the performance slightly in the HF-Be256 case on Hopper. The third application does not seem to have much impact on Hopper, but Titan and Intrepid require more applications of the load balancer before HF-Be512 converges.

Figure 10 shows the processor utilization over time for two HF-Be256 runs on 2400 and 9600 cores of Hopper. The graph demonstrates that for the first iteration of the computation, before retentive stealing is applied, performance is low due to many cores spending time trying to steal tasks. As the problem is strong scaled, the amount of time spent stealing increases dramatically. Retentive work stealing reduces this time in both cases, increasing the amount of time spent executing tasks. With retention, work stealing performs almost as well as persistence-based load balancing.

8. CONCLUSION

We presented scalable algorithms for persistence-based load balancing and work stealing. The hierarchical persistence-based load balancing algorithm presented locally rebalances the load in a greedy fashion. The work stealing algorithm is optimized for distributed memory machines, by coalescing remote operations, reducing the duration of locked opera-

tions, and enabling concurrent steal operations that allow overlapped task migration. We presented retentive stealing to further adapt work stealing for iterative applications.

Work stealing is traditionally considered not to scale beyond small core counts, due to its perceived limitations: randomization, the need to repeatedly rebalance, the cost of termination detection, and the potential interference with application execution. While not universally applicable, we demonstrate that work stealing scales better than commonly believed. Retentive stealing is also shown to improve execution efficiency by incrementally improving the load balance and reducing the overheads associated with stealing.

ACKNOWLEDGEMENTS

This work was supported in part by the DOE Office of Science, Advanced Scientific Computing Research, under award number 59193, and by the U.S. Department of Energy's Pacific Northwest National Laboratory under the Extreme Scale Computing Initiative. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, the National Energy Research Scientific Computing Center and the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which are supported by the Office of Science of the U.S. Department of Energy under contracts DE-AC02-06CH11357, DE-AC02-05CH11231, and DE-AC05-00OR22725, respectively.

9. REFERENCES

[1] ALCF Intrepid. `http://www.alcf.anl.gov/intrepid`.

[2] NERSC Hopper. `http://www.nersc.gov/users/computational-systems/hopper`.

[3] OLCF Titan. `http://www.olcf.ornl.gov/computing-resources/titan`.

[4] G. Baumgartner et al. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proc. of IEEE*, 93(2):276–292, 2005.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*, pages 207–216, July 1995.

[6] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX*, pages 10–10, 1997.

[7] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):673–693, 1999.

[8] Ü. V. Çatalyürek, E. G. Boman, K. D. Devine, D. Bozdag, R. T. Heaphy, and L. A. Riesen. A repartitioning hypergraph model for dynamic load balancing. *JPDC*, 69(8):711–724, 2009.

[9] A. Chandramowlishwaran, K. Knobe, and R. Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *IPDPS*, 2010.

[10] P. Charles, C. Grothoff, V. Saraswat, et al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[11] N. H. Darach Golden and S. McGrath. Parallel adaptive mesh refinement for large eddy simulation using the finite element methods. In *PARA*, pages 172–181, 1998.

[12] A. Darte, J. Mellor-Crummey, R. Fowler, and D. C. Miranda. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. *JPDC*, 63(9):887–911, 2003.

[13] R. Das, Y.-S. Hwang, M. Uysal, J. Saltz, and A. Sussman. Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics. In *Scalable Parallel Libraries Conference*, pages 45 –56, oct 1993.

[14] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *SC*, 2009.

[15] N. Francez. Distributed termination. *ACM Trans. Program. Lang. Syst.*, 2:42–55, January 1980.

[16] E. Gabriel et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European PVM/MPI*, September 2004.

[17] G. R. Gao, T. L. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *IPDPS*, pages 1–6, 2007.

[18] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.*, 16:452–469, March 1995.

[19] C. Joerg and B. C. Kuszmaul. Massively parallel chess. In *Proceedings of the Third DIMACS Parallel Implementation Challenge, Rutgers*, 1994.

[20] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *OOPSLA'93*, pages 91–108, September 1993.

[21] G. Karypis, K. Schloegel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[22] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The ARMCI approach. *Int. J. High Perform. Comput. Appl.*, 20(2):233–253, 2006.

[23] J. C. Phillips et al. Scalable molecular dynamics with namd. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[24] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[25] V. A. Saraswat, P. Kambadur, S. B. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *PPoPP*, pages 201–212, 2011.

[26] A. Szabo and N. S. Ostlund. *Modern Quantum Chemistry*. McGraw-Hill Inc., New York, 1996.

[27] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *PPoPP*, pages 34–43, 2001.

[28] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Pract. Exper.*, 3:457–481, October 1991.

[29] G. Zheng, A. Bhatele, E. Meneses, and L. V. Kale. Periodic Hierarchical Load Balancing for Large Supercomputers. *IJHPCA*, 2010.