CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3747

Work stealing for GPU-accelerated parallel programs in a global address space framework

Humayun Arafat[1,*,†], James Dinan[2], Sriram Krishnamoorthy[3], Pavan Balaji[2] and P. Sadayappan[1]

[1]*Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA*
[2]*Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA*
[3]*Computer Science and Mathematics Division, Pacific Northwest National Laboratory, Richland, WA, USA*

SUMMARY

Task parallelism is an attractive approach to automatically load balance the computation in a parallel system and adapt to dynamism exhibited by parallel systems. Exploiting task parallelism through work stealing has been extensively studied in shared and distributed-memory contexts. In this paper, we study the design of a system that uses work stealing for dynamic load balancing of task-parallel programs executed on hybrid distributed-memory CPU-graphics processing unit (GPU) systems in a global-address space framework. We take into account the unique nature of the accelerator model employed by GPUs, the significant performance difference between GPU and CPU execution as a function of problem size, and the distinct CPU and GPU memory domains. We consider various alternatives in designing a distributed work stealing algorithm for CPU-GPU systems, while taking into account the impact of task distribution and data movement overheads. These strategies are evaluated using microbenchmarks that capture various execution configurations as well as the state-of-the-art CCSD(T) application module from the computational chemistry domain. Copyright © 2015 John Wiley & Sons, Ltd.

Received 11 March 2013; Revised 30 January 2015; Accepted 22 November 2015

KEY WORDS: GPU; partitioned global address space; task parallelism

1. INTRODUCTION

Computer systems, from desktops to high-end supercomputers, are being constructed from an increasing number of processor cores. As semiconductor feature sizes decrease, power limits [1] and resilience [2] are becoming primary concerns for efficiently exploiting the available hardware resources. In addition, system noise [3] will play an increasingly important role as hardware parallelism grows. These factors result in dynamic variation in the hardware resources and functionality available at any given point in time. Static partitioning of work to utilize such a dynamic system is expected to be infeasible.

Task parallelism therefore is increasingly being considered as a solution to this problem. The work to be performed is divided into finer-grained units, referred to as *tasks*. While several abstractions for task parallelism have been considered, we focus on task collections scheduled by using work stealing. The programmer specifies the concurrency and dependences inherent in the program. This specification is then dynamically mapped onto the processor cores, dynamically reacting to system changes.

Work stealing has been extensively studied on shared-memory and distributed-memory systems. In this paper, we explore the design space in constructing a framework for work stealing task-parallel

*Correspondence to: Humayun Arafat, Department of Computer Science and Engineering, The Ohio State University.
†E-mail: arafat.6@osu.edu

collections on heterogeneous CPU-graphics processing unit (GPU) systems. To address the key challenges in utilizing accelerator systems, we introduce Scioto-ACC. The Scioto-ACC framework provides the programmer with a highly productive, task-parallel programming interface where sections of the computation are executed during task-parallel phases. A global address space is used to store data operated on by tasks, enabling processing of big datasets and permitting the runtime system to automatically balance the workload across nodes. Scioto-ACC automatically dispatches tasks to CPU cores and GPU accelerators based on resource availability and suitability of the computation to the given compute device. Scioto-ACC uses an extended accelerator-aware work-stealing algorithm to perform scalable, receiver-initiated, dynamic load balancing.

Graphics processing units (GPUs) have traditionally been treated as accelerators, with one CPU (process or thread) on each compute node driving the associated GPU. We observe that such an approach does not lead to the most efficient execution. We consider various alternatives to a scalable distributed-memory work stealing that are better suited to CPU-GPU hybrid systems. We also evaluate various strategies for distributing tasks among the compute nodes in order to better exploit data locality. Managing data movement between the disjoint CPU and GPU memory domains is as important as scheduling computation on hybrid systems. Our work-stealing strategies consider pipelining of communication from the global address space to the GPU memory as an integral part of scheduling.

The various design choices are evaluated by using a microbenchmark that can be tuned to comprise different numbers of CPU-favorable and GPU-favorable tasks. The schemes are also evaluated by using a coupled cluster module in NWChem, called CCSD(T), an application naturally structured to exploit task parallelism. We observe that the work-stealing design employed for CPU-only systems performs poorly as compared to the alternatives that take heterogeneity into account.

2. BACKGROUND AND RELATED WORK

In this section, we discuss and provide background information on the context of this work and discuss prior relevant related work.

Global Address Space Programming Models. Global address space programming models [4–7] allow the construction of global data structures across the memory domains of a distributed system, while allowing individual processes (or threads) to access arbitrary portions of the data structure independently. This provides an attractive substrate to support task parallelism on distributed-memory systems where each task can be executed by any process with its inputs and outputs located in the global address space [8]. For example, UPC [9, 10] and Global Arrays (GA) [11] have been extended to support task parallelism. Compiler-based task parallelism and hybrid computation were studied in [12].

Global Arrays. We demonstrate our framework using GA. GA [13, 14] is a Partitioned Global Address Space (PGAS) library that provides support for globally addressable multidimensional arrays on distributed-memory platforms. It is a portable library available on most computing platforms and is employed in several applications, including NWChem [15], MOLPRO [16], and ScalaBLAST [17]. Using GA, data buffers allocated on distinct processes are viewed as a multidimensional array, with interfaces for any process to asynchronously access an arbitrary multidimensional sub-patch of the array. GA also exposes locality information and direct access to the underlying data buffers to enable further optimizations.

Global Arrays (GA) provides one-sided put, get, and accumulate operations. One-sided put operations copy data from the local buffer to the global array sections; get operations copy data from global array section to a local buffer; and accumulate operations reduce data from a local buffer into a section of the global array.

Work Stealing and CPU-GPU Co-scheduling. Work stealing has been extensively studied as a means for balancing the workload across nodes, as well as within CPU and GPU nodes [18–22].

Ravi *et al*. [23] addressed the effective partitioning and scheduling of generalized reductions across the cores of a multicore CPU and attached GPU. The Habanero project has investigated mapping of the X10 async-finish task execution model to GPU accelerators [24]. Compiler-based

DOI: 10.1002/cpe

solutions have also been developed to address task parallelism in distributed-memory systems [25–28]. Global load balancing was studied in [29] in the context of the high productivity language, X10. This work addresses load balancing for irregular problems, as well as communication optimization and termination detection. While there are similarities in the approach to task parallelism, our work is performed in the context of the GA PGAS programming model.

CUDASA [30] extends the CUDA [31] programming system to clusters of GPUs. Virtual OpenCL [32] provides a layer of virtualization between OpenCL applications and GPUs, enabling transparent mapping of virtual GPUs to clusters of CPU and GPU hardware, as well as enabling dynamic load balancing through virtual GPU migration. OmpSs [33] is a variation of the OpenMP extended to support asynchronous task parallelism in a heterogeneous cluster. StarPU [34] provides a unified programming interface and runtime system for heterogeneous systems. Other approaches include static and dynamic assignment of tasks and functions to CPUs and GPUs based on data placement [35, 36] support for task graph scheduling [37–39], reductions [23, 40, 41], and scheduling of processing resources across kernels [42]. These approaches did not consider the scheduling of task-parallel programs using work stealing on hybrid CPU-GPU systems.

Scalable Collections of Task Objects. Scalable collections of task objects (Scioto) [8, 11] defines a task-parallel execution model in which tasks operate on data in a globally accessible data store. In this work, we extend Scioto to provide support for both CPU and accelerator execution of tasks. In the Scioto model, the user generates task objects into a task collection, which is shared across a group of processes; tasks in the collection are executed in task-parallel phases. Scioto task objects contain a portable reference to a task execution function that is available on all processes participating in the task collection, as well as user-supplied task arguments that can reference locations in the global data store. Because Scioto tasks operate on globally available data, they can be executed at any location, enabling the runtime system to perform automatic scheduling and load balancing to optimize communication involved in task execution.

Scioto utilizes the GA PGAS model and builds a scalable runtime system built on top of ARMCI's one-sided communication model [43]. This runtime system creates shared task queues, which are split to allow lock-free local access to a reserved segment of the queue and remote, one-sided access to a shared segment of the queue. Load balancing is performed by using work stealing, where processes with no work lock the shared queue segment at a randomly selected victim and transfer a portion of the victim's work to the thief's queue.

3. FRAMEWORK DESIGN

In this section, we describe the Scioto-ACC framework and the various design choices considered in dynamic load-balancing task-parallel collections on heterogeneous CPU-GPU systems.

Figure 1 illustrates the distributed-memory system of interest in this work: a CPU-GPU cluster connected by a high-performance interconnection network. The total available memory on each node is partitioned into CPU memory, which is efficiently accessible to the multicore CPUs, and GPU memory, which is on the same end of the PCI express bus as the GPUs. Data is distributed across the nodes of the cluster by using GA, aggregating the distributed CPU memories of the compute nodes. Communication between different nodes is handled by GA one-sided communication routines. Each CPU can communicate with its GPU through the PCI express bus by using synchronous and asynchronous CUDA API calls.

3.1. Design overview

The Scioto task-parallel library maintains a local queue of tasks with each process. If all tasks in the local queue of a process have been processed, tasks from other process queues are sought by using work stealing. In this paper, we extend this model for heterogeneous systems. The extended system enables transparent execution of tasks by using the cores of the hosts as well as the GPUs in a cluster. The programmer needs to specify only the tasks using the interface provided. The library handles the load balancing and scheduling across nodes as well as between CPU cores and GPUs. The task API (details provided later in this section) is similar to that of the previously developed

DOI: 10.1002/cpe

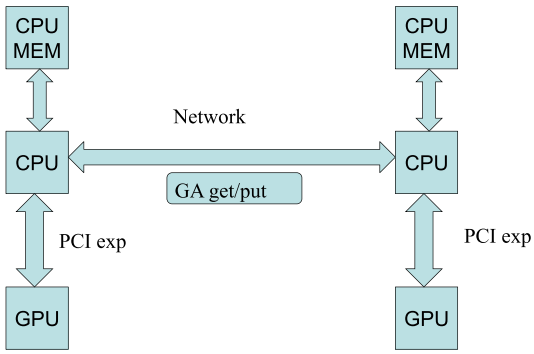


Figure 1. Conceptual architecture of the hybrid CPU-graphics processing unit system.

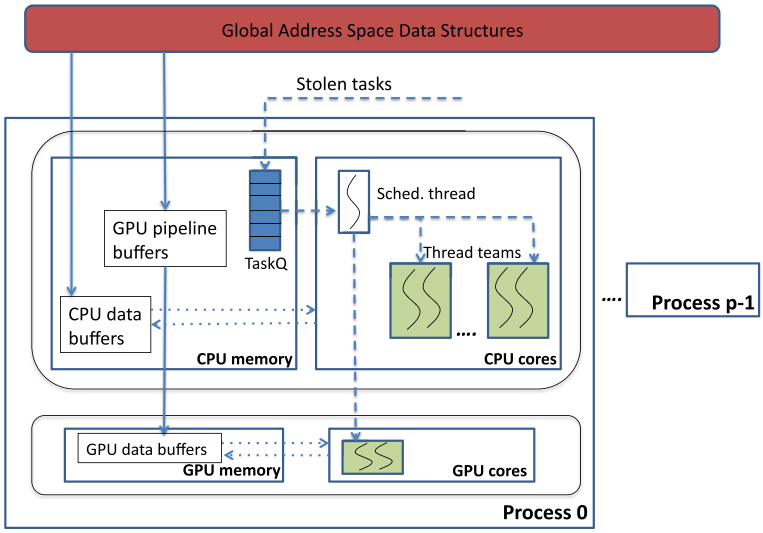


Figure 2. Scioto-ACC framework for scheduling tasks across CPUs and graphics processing units (GPUs). Solid arrows denote pipelined data transfer operations. Dashed arrows denote task-scheduling actions. Dotted arrows represent efficient (load-store) access to local data during task execution.

Scioto system, except that the API specifies the GPU kernels to execute tasks and a user function to provide information to the system about the estimated ratio of execution time of a task instance on the GPU versus CPU.

Figure 2 depicts the structure and flow within each node in the Scioto-ACC system. Each compute node contains a scheduler that manages the execution of tasks as well as remote memory operations. The scheduler orchestrates the asynchronous fetching of input data for tasks from remote nodes, execution of tasks on CPU thread teams and attached GPU, and transmission of output results of tasks to remote nodes. The processing elements are organized into CPU thread teams and the GPUs. The scheduler dispatches the tasks to the thread teams and the GPUs, taking care of any associated communication. The scheduler is also responsible for stealing work from other compute nodes to maximize processor utilization. In the figure, PGAS memory is shown in red. Each process can read and write from the global address space. Buffers employed in managing data transfer are allocated in the CPU and GPU memories. Each member of the thread team can be multithreaded or single threaded based on the application.

The execution of each task is divided into three parts: fetching the input data, executing the tasks, and updating the output data. The scheduler in each node takes a chunk of tasks from its queue and issues the input operations for those tasks. This data transfer is done asynchronously with a

DOI: 10.1002/cpe

nonblocking data transfer, using pinned memory. When the data transfer for all operands to a task completes, the task is scheduled for execution either on a CPU thread team or on a GPU. If a task is to be executed on a GPU, additional steps are taken to transfer the data to the GPU memory. When a task completes, the scheduler issues an update operation to reflect the changes to the data in the global array using nonblocking operations.

The scheduler thus enables the overlap of computation on the CPU cores and GPUs with data transfer across cluster nodes as well as between CPU memory and GPU memory within the nodes.

3.2. Task interface

The Scioto-ACC interface consists of the following parts:

- Input and output: Task inputs and outputs are specified in terms of global array handles and index ranges that correspond to portions of data in the global arrays.
- Task descriptor: Function descriptors for each type of processing element – CPU and GPU.

The typical approach to using the Scioto-ACC framework is illustrated by Algorithm 1.

```
begin
    tasks ⟵ ∅
    while hasMoreInputTasks do
        T ⟵ createTask()
        tasks ⟵ insertTask(T)
    end
    processTasks()
end
```

Algorithm 1: Insertion of tasks into the system

The work to be performed is specified in terms of tasks. Tasks are inserted into the runtime system by using the provided interface and are processed by invoking the processTasks function. When this function returns, all tasks have been processed, and the results are reflected in the GA specified as outputs in the tasks.

Figure 3 shows sample C source code to illustrate the interface provided by the library. Here, the user provides four functions that define the behavior of a task. The *task_in* and *task_out* functions read and write a task's inputs and output to and from the global address space. The *task_cpu_fcn* and *task_gpu_fcn* can be used to execute the task on the CPU or GPU, respectively. The *task_exec_est* function provides a performance model for the given task that can be used to determine whether it should be executed on the CPU or the GPU. MPI and GA are first initialized in the main method. The task's functions are registered by using *tc_register*. A Scioto task collection is created by using the *tc_create* function, and tasks are then inserted in the collection by using *tc_add*. The collective function *tc_process* causes execution of the tasks in the task collection. Finally, *tc_finalize* is invoked to clean up memory.

Figure 4 provides an overview of the pipelined task scheduler's operation. The job of this algorithm is to retrieve CPU and GPU tasks as needed, and direct their execution through the data movement and execution stages, shown in Figure 6. The outer loop checks for global termination, which can only be achieved when *MarkCompleted* has been called for every task in the system. Within this loop, the scheduler issues new tasks when execution slots are available and attempts to move in-flight tasks that have completed a particular stage of execution into the next stage.

In the first step, the scheduler checks if CPU or GPU task execution slots are available. If they are, the scheduler retrieves the corresponding task type and initiates its execution; the task is tagged with the type of execution selected by the scheduler, which will later be used to guide it through the appropriate execution stages. When retrieving a task, if none are available locally, the scheduler

DOI: 10.1002/cpe

```
typedef struct {
 ....
} user_task;

void task_cpu_fcn(tc_t tc, task_t *task) {
  user_task *utask = tc_task_body(task);
  // Perform execution of a task in CPU
}
void task_gpu_fcn(tc_t tc, task_t *task) {
  user_task *utask = tc_task_body(task);
  // Perform execution of a task in GPU
}

void task_in(tc_t tc, task_t *task) {
  user_task *utask = tc_task_body(task);
  // Perform input Get operation from GA
}
void task_out(tc_t tc, task_t *task) {
  user_task *utask = tc_task_body(task);
  // Perform output Accumulate operation in GA
}

int task_exec_est(tc_t tc, task_t *task) {
  // Estimate execution time on the accelerator versus CPU
  // Values: < 1, CPU is better; 1 == Same; > 1 GPU is better
}

void main(int argc, char **argv) {
  tc_t tc; task_handle_t *hdl;
  task_t *task; user_task *utask;
  // Initialize MPI and Global Arrays

  tc_init(&argc, &argv);

  tc = tc_create(sizeof(user_task), CHUNK_SIZE, MAX_TASKS);
  hdl = tc_register(tc, task_cpu_fcn, task_gpu_fcn, task_in, task_out, task_exec_est);
  task = tc_task_create(sizeof(user_task), hdl);
  utask = tc_task_body(task);

  while(/*hasMoretasks*/) {
    setup_task(utask); /* set input and output arguments in the task body */
    tc_add(tc, utask, AFFINITY_HIGH, task);
  }

  tc_process(tc);
  tc_destroy(tc);
  tc_finalize();

  // Finalize Global Arrays and MPI
}
```

Figure 3. Illustration of task-parallel execution using the Scioto-ACC framework.

will perform a non-blocking steal attempt to acquire more work. If work is found, a task will be returned. If not, execution proceeds and stealing will be attempted again on the next iteration.

In our model, all tasks read inputs from and write outputs to global arrays; additional task metadata could be added to allow for tasks that perform a subset of these actions. Non-blocking communication is performed to overlap data movement with execution and enable pipelined parallelism. GA provides only a blocking *GA_NbWait* operation for completing asynchronous communication. Thus, to avoid blocking on all tasks performing GA communication, we select the oldest task, from the head of the corresponding pipeline state queue and wait for its communication to complete.

Data movement between the CPU and GPU memories is performed using asynchronous, non-blocking copy operations. When advancing tasks through these stages, we test for the completion of each operation and advance those that have completed. Next, tasks are launched for execution. For GPU tasks, we dispatch the user's kernel to the GPU device. For CPU tasks, we dispatch it to a thread waiting to execute the given operation. Oversubscription of the node can be avoided in this stage by queueing tasks when cores are unavailable and launching new tasks as others complete.

Once tasks have finished execution, results are copied back to CPU memory if needed and then are published to the global array. If the local process has become idle, it participates in termination detection. If the computation has not quiesced, the scheduler resumes activity and attempts to acquire tasks through work stealing.

We now discuss each of the part of the system and its implementation in our framework.

3.3. Work stealing and task scheduling

The scheduler on each machine takes the task from the queue and runs it on the CPU and GPU based on their position on the queue. If the CPU is idle, the scheduler dequeues chunk of tasks

DOI: 10.1002/cpe

```
Constants: MaxGPUTasks, MaxCPUTasks
Pipeline Queues: GAFetchQ, GPUWriteQ, GPUExecQ,
     CPUExecQ, GPUReadQ, GAUpdateQ

while (! terminated ) {
    /* Inject GPU tasks into the pipeline */
    for ( ; GPUTasks < MaxGPUTasks ; GPUTasks++ ) {
        T = FetchGPUTask(tc) /* Steal if needed */
        T.type = GPU
        GAFetchQ = GAFetchQ + T
        FetchGA(T)
    }

    /* Inject CPU tasks into the pipeline */
    for ( ; CPUTasks < MaxCPUTasks ; CPUTasks++ ) {
        T = FetchCPUTask(tc) /* Steal if needed */
        T.type = CPU
        GAFetchQ = GAFetchQ + T
        FetchGA(T)
    }

    /* Advance the oldest GA communcation */
    if (GAFetchQ != nil) {
        T = CompleteGAFetch(head(GAFetchQ))
        if (T.type == GPU) {
            GPUWriteQ = GPUWriteQ + T
            CopyAsyncCPUtoGPU(T)
        } else {
            CPUExecQ = CPUExecQ + T
            SpawnCPU(T)
        }
    }

    /* Check for completed CPU->GPU transfers */
    foreach ( T in GPUWriteQ ) {
        if (CopyAsyncCPUtoGPUCompleted(T)) {
            GPUWriteQ = GPUWriteQ - T
            GPUExecQ = GPUExecQ + T
            SpawnGPU(T)
        }
    }

    /* Check for completed GPU tasks */
    foreach ( T in GPUExecQ ) {
        if (GPUExecCompleted(T)) {
            GPUExecQ = GPUExecQ - T
            GPUReadQ = GPUReadQ + T
            CopyAsyncGPUtoCPU(T)
        }
    }

    /* Check for completed CPU tasks */
    foreach ( T in CPUExecQ ) {
        if (CPUExecCompleted(T)) {
            CPUExecQ = CPUExecQ - T
            GAUpdateQ = GAUpdateQ + T
            UpdateGA(T)
        }
    }

    /* Check for completed GPU->CPU transfers */
    foreach ( T in GPUReadQ ) {
        if (GPUExecCompleted(T)) {
            GPUExecQ = GPUExecQ - T
            GAUpdateQ = GAUpdateQ + T
            UpdateGA(T)
        }
    }

    /* Check for completed GA transfers */
    foreach ( T in GPUReadQ ) {
        if (GPUExecCompleted(T)) {
            GPUExecQ = GPUExecQ - T
            GAUpdateQ = GAUpdateQ + T
            UpdateGA(T)
        }
    }

    /* Advance the oldest GA communcation */
    if (GAUpdateQ != nil) {
        T = CompleteGAUpdate(head(GAUpdateQ))
        MarkCompleted(T)
        if (T.type == GPU) GPUTasks--
        else CPUTasks--
    }

    if (GPUTasks == 0 && CPUTasks == 0)
        terminated = CheckForTermination()
}
```

Figure 4. Pipelined task-scheduling algorithm pseudocode.

from the front of the queue and pushes them into the CPU pipeline. If, on the other hand, the GPU pipeline is empty, the scheduler takes tasks from the back of the queue and pushes them into the pipeline of GPU. When the queue is empty, the scheduler will perform work stealing in order to get free tasks available for execution. This work stealing is performed by using remote direct memory access from the distributed-memory system. A CPU-only distributed system differs significantly from a heterogeneous system. Scioto can use different algorithms when stealing work from victim processes. We show that different work-stealing approaches are beneficial when using hybrid systems, compared with the direct extension of the systems developed for homogeneous clusters.

Our baseline design is shown in Figure 5. In this design, the queue is sorted according to the performance model values generated by *task_exec_est*. Here, the front of the queue contains the tasks that take less time on the CPU than on the GPU. On the other hand, the end of the queue has GPU-friendly tasks. When the task queue is empty, a process selects another process at random as a victim, where also a similar queue is maintained.

When all available tasks are completed, the runtime system goes into a collective termination detection phase to determine if no tasks exist anywhere in the system. If this is confirmed, the program terminates.

To better support hybrid CPU-GPU systems, we consider the following task queue configurations and work stealing choices.

Single-queue, single-ended steal. The original CPU-only Scioto programming model uses a single-queue and single-ended steal. The tasks are inserted in the queue before the beginning of the execution. They can also be created dynamically and then inserted at one end of the queue. After

DOI: 10.1002/cpe

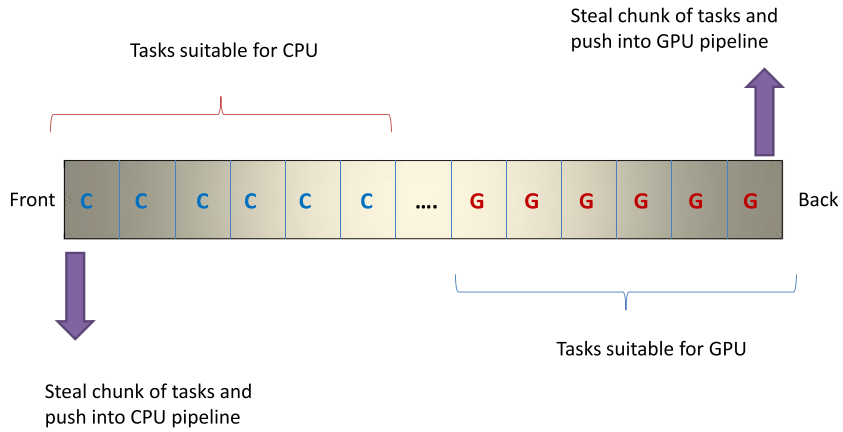


Figure 5. Double-ended queue organized as tasks suited for CPU and graphics processing unit (GPU) execution.

creating all tasks, the user calls a function gtc_process, which starts the execution of tasks from the queue. When the queue is empty, the Scioto library steals tasks from a remote or local process. This stealing operation takes tasks from the other end of the queue. As soon as the library is finished running the tasks, it goes to termination detection, and the program terminates.

Double-queue, single-ended steal. In this scheme, we discriminate between tasks. That is, each task can run on either the CPU or GPU. Running time for each task varies based on the selection of hardware. For this reason, a natural extension of the previous work is to include another queue for GPU tasks. The user will add tasks to a queue as desired. If the running time of a task is faster, the task is added to the GPU queue; if the time is slower, the task is added to the CPU queue. The stealing operation is similar to that in the Scioto programming model.

Single-queue, double-ended steal. The double-queue strategy can result in higher overheads, especially when the tasks are not properly load balanced initially. The number of steal operations has increased because of maintaining two different queues. In the next work-stealing strategy we studied, therefore, a single queue is used, but the stealing operation now steals from both ends of the queue. At the beginning of the task-parallel phase, the user gives some information about the CPU and GPU running time for each task. Based on the ratio of CPU time and GPU time, the library sorts the tasks in its local queue. In this approach, the CPU tasks reside at the beginning of the queue, and GPU tasks are placed at the end of queue; tasks that are suitable for both CPU and GPU stay in the middle. The library dequeues the tasks from the front of the queue and schedules them to run on a CPU; it also takes tasks from the other end of the queue and executes them on the GPU. In this way, if the tasks are load balanced, the tasks are executed where they achieve better performance. Operation is completed, we have both CPU-friendly and GPU-friendly jobs in the queue. This approach also helps reduce the number of steals in the system.

Single-queue, CPU/GPU-ended steal. The previous scheme involves stealing work from other processes when that process has finished executing all tasks in its local queue. However, such a scheme can enforce some tasks to run on the GPU when it is better to run them on the CPU and vice versa. For this reason, in this model, we explore different techniques based on the ratio of CPU time and GPU time for each task. While running tasks from the CPU end of the queue, the ratio can change to more than one, indicating that the queue no longer has any CPU-friendly tasks and hence that it would not be beneficial to run the tasks on CPU. The same situation can happen in the context of GPU tasks. The single-queue, single-ended steal algorithm is developed to handle this situation. Here, the library steals immediately when it runs out of any type of tasks. We explored three scenarios where the library steals CPU tasks, steals GPU tasks, and steals both tasks. From our experiment, stealing GPU tasks was observed to be the most beneficial.

DOI: 10.1002/cpe

3.4. Data management

Scioto-ACC enables transparent execution of task-parallel computations using both CPU cores and GPUs in a cluster, using a global address space. Tasks are specific jobs that can be run either on the CPU or the GPU. Data movement occurs in three layers: global address space, host memory, and device memory. Initially, the input data for one task can be distributed in the global address space. Data is moved to the host memory for the execution of the task. If the scheduler decides that the task is to be executed on the host, then the function provided by the user for that task is executed. After the task has finished executing, the output is used to update a specified memory location in the global address space. This is a one-sided operation that can also be pipelined. The data transfer is transparently handled by the Scioto-ACC system using nonblocking pipelined transfer. Therefore, more than one task can be concurrently active.

In order to execute a task on a GPU, the input data must be transferred to the global memory of GPU; after the task is finished, the output data should be copied back to host memory. Scioto-ACC handles this process automatically by using pipelining. The input data is first brought from the global address space to local host memory and transferred to the global memory of GPU; the kernel then is launched for execution on the GPU. This data transfer is also handled automatically by Scioto-ACC in a pipelined manner. After the task completes, the output data is transferred from the device memory to host memory, and the global address space is updated. This is also a one-sided communication and can be pipelined.

Figure 6 shows different pipeline buffers required in the system. Host pipeline buffers are required in order to transfer data to and from global address space. Again, device pipeline buffers are required in order to transfer data between host and device. In this section, we discuss buffer management both for host and device.

Buffer management. Pipelining for the global address space is implemented by using the nonblocking calls of GA. It is essential to use nonblocking calls to ensure pipelining in our system. Scioto-ACC has multiple levels of buffers that can help hide the communication cost.

Global Arrays (GA) provide nonblocking get and nonblocking put functions, which can be used to perform one-sided operations. We use *GA_NbGet*, *GA_NbPut*, and *GA_NbAcc* for nonblocking operations. Each takes the index of the GA as input and performs the respective operations.

Data must be transferred to the device if the task scheduler decides to run the data on the GPU. The same buffering mechanism for PGAS is also useful for the GPU. The pipelining mechanism for the GPU is discussed below. However, one important issue to consider is the amount of memory. In some cases, memory requirement is high for the CCSD(T) application (discussed later). Increasing the pipeline levels without taking memory limitation into consideration can quickly fill the GPU memory.

GPU pipelining. CUDA provides asynchronous operations for overlapping communication and computation. *cudaMemcpy* is used to transfer data between host and device in a blocking fashion. When this function returns to the host process, the transfer of the data is already finished. CUDA also provides a nonblocking version of memory copy called *cudaMemcpyAsync*. This function returns immediately to the host process. However, the asynchronous copy on the GPU needs the local buffer as pinned memory. Pinned or page-locked memory can be obtained by using the runtime function *cudaHostAlloc*. Pinned memory can give much higher bandwidth than the memory allocated by *malloc*, which returns pageable memory. Too much page-locked memory, however, can degrade the performance of the overall system. For this reason, it should be carefully used. Typically, it should be used for data transfer between host and device. CUDA also provides another function, called

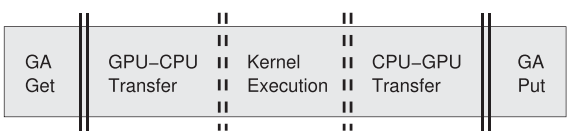


Figure 6. Pipelining between global address space, CPU, and graphics processing unit (GPU) memory domains; host pipeline buffers are shown as solid lines, and GPU pipeline buffers are shown as dashed lines.

DOI: 10.1002/cpe

cudaHostRegister, that can page-lock previously allocated memory by using another library. The memory will behave similar to a memory returned by *cudaHostAlloc*. Nonblocking data transfer operations can be performed with a registered memory. This is important for those systems where CPUs and GPUs are used for computation and where data transfer is required between the host and device. Our library can allocate pinned memory and use it for global address space and GPU programming environments for efficient asynchronous operations.

4. EXPERIMENTAL EVALUATION

We next discuss the experimental evaluation of the effectiveness of the Scioto-ACC system. We report on results on a GPU cluster for both a parameterized microbenchmark and a state-of-the-art computational chemistry method for ab initio modeling of properties of materials.

4.1. Experimental setup

The Scioto-ACC framework was experimentally evaluated on a GPU cluster with 32 16-core nodes, each consisting of an Intel Xeon CPU E5520 (2.27 GHz) with 48 GB of memory per node and an NVIDIA Tesla C2050 GPU. The system is interconnected by using InfiniBand. MVAPICH2 was used for MPI, and all programs were compiled by using the Intel Compiler version 12 [44] and CUDA 5.0.

A benchmark derived from a compute-intensive quantum chemistry application, perturbative triples correction in the coupled cluster singles and doubles method (CCSD(T)) [45], was used to evaluate the effectiveness of the Scioto-ACC framework. CCSD(T) represents one of the most accurate computational methods for ab initio modeling of the electronic properties of materials and is implemented in most modern quantum chemistry software suites, such as NWChem, ACES, MOLPRO, Gaussian, and QCHEM. We used a recently developed version of the Tensor Contraction Engine [46] to generate the set of tasks for the CCSD(T) computation. Starting from an abstract high-level description of the computation as a collection of mathematical tensor expressions, the Tensor Contraction Engine automatically synthesizes concrete parallel code for the computation, in the form of a set of tasks and the intertask dependencies. The total work is structured as a sequence of steps, with a large number of independent tasks in each step and barrier synchronization between the steps.

The most compute-intensive kernel at the core of all coupled cluster codes is a tensor contraction, essentially a generalized multidimensional matrix product. Tensor contractions are typically implemented by a layout permutation, if needed, followed by invocation of DGEMM implemented in highly optimized vendor libraries. Because matrix–matrix multiplication constitutes the core computation, we first discuss the performance of matrix multiplication on a GPU compared with its execution on the cores of the host CPU.

4.2. Matrix multiplication

Coupled cluster methods use matrix multiplication extensively—it represents the computationally dominant component of the calculation. Figure 7 shows the time for matrix–matrix multiplication on the CPU versus GPU, as a function of matrix size, The blue line shows the time for DGEMM operations with the Intel Math Kernel Library, using four threads. The red curve shows the DGEMM kernel execution time on the GPU. For matrix sizes less than 800, it is faster to perform the multiplication on CPU cores, whereas execution on the GPU is faster for larger matrices. For small matrices, the data transfer time between CPU and GPU dominates the computation time on the GPU. On the other hand, for large matrices, the computation time is much larger than data movement time, and the higher computational rate on the GPU makes it faster even after accounting for the CPU-GPU data transfer time.

This data shows that execution of different matrix-multiplication instances on different processing elements—CPU or GPU—is necessary for optimizing performance. The Scioto-ACC library aids in the effective use of the CPU cores and GPUs by identifying the tasks better suited to each.

DOI: 10.1002/cpe

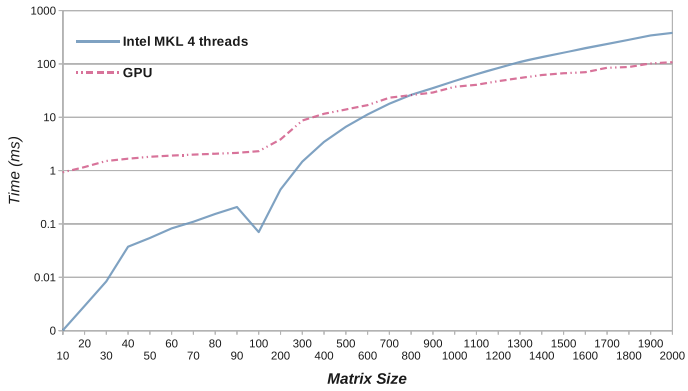
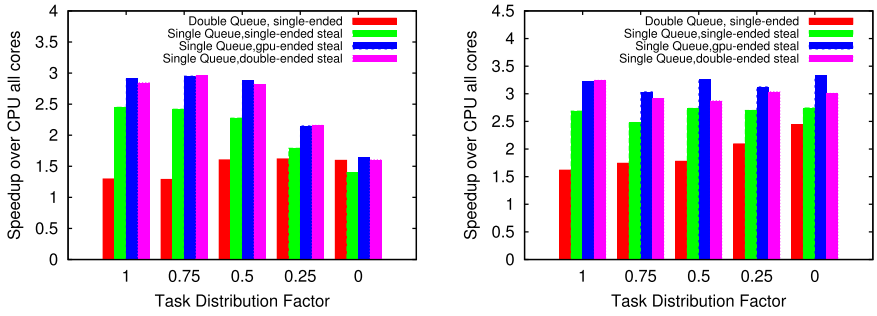
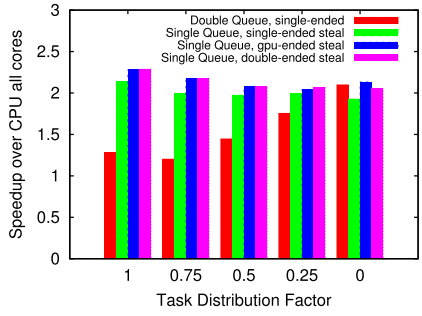


Figure 7. Comparison of execution times for matrix multiplication time on NVIDIA Tesla C2050 GPU and MKL four-thread parallel execution on Intel Xeon CPU.



(a) Ratio of GPU and CPU work = 1

(b) Ratio of GPU and CPU work = 2



(c) Ratio of GPU and CPU work= 0.5

Figure 8. Comparison of different work-stealing algorithms using varied tasks. Task Distribution Factor 1 indicates highest skewed task distribution where one process has all tasks.

Figure 8 shows experimental performance data for a microbenchmark comprising a collection of matrix–matrix multiplication tasks of varying matrix sizes. The different work-stealing strategies were compared for different task mixes, representing different fractions of work in GPU-favored tasks versus CPU-favored tasks, as well as different degrees of skew in the initial task distributions at the different nodes of the cluster. Figure 8(a) presents results for equal amounts of work (number of arithmetic operations) across CPU-favored and GPU-favored tasks. The number of CPU tasks are higher compared with GPU task. The reason is to keep the total number of operations within the ratio. In this experiment, we have used 10×10 sized matrices for CPU tasks and 1000×1000 for GPU tasks. The number of tasks are varied to match the ratio. The five sets of data correspond to varying degrees of skew in the initial distribution of the tasks. For all cases, performance is reported relative to CPU-only execution using the original Scioto implementation [43]. The rightmost set of bars shows results for the case of no skew', that is, each cluster node has the same ratio of

DOI: 10.1002/cpe

GPU-favored and CPU-favored jobs. This therefore represents a perfectly balanced initial distribution of tasks among all nodes, and there is little difference in performance between the four work-stealing variants. As we move from the right to the left, the degree of skew increases.

At the left end of the graph, the skew is maximal, representing extreme imbalance, wherein all tasks are initially on a single node and all other nodes have empty task lists. For intermediate values of skew s, the fraction $1-s$ of jobs is distributed in a balanced manner among that fraction of nodes, while the fraction s is placed in a single other node. For example, with a skew of 0.25 on the 32 node cluster, 75% of the load is uniformly distributed over 24 nodes, while 25% of the load is initially placed on one of the remaining 8 nodes, and 7 nodes have no initial work.

As the skew increases, there is increased work-stealing activity during execution, for both the baseline CPU-only case and the hybrid cases. The double-queue scheme (leftmost bar, colored red) requires work-stealing activity for both its queues, and therefore, the relative speedup over the baseline CPU-only (original Scioto scheme) gets worse with higher skew. But for the single-queue work-stealing schemes, performance improves relative to the CPU-only case as the skew increases. With all of them, each node maintains a single sorted queue of tasks, with CPU-favored tasks at one end and GPU-favored tasks at the other end. With the single-ended work-stealing scheme (colored green, second bar from the left), work stealing is initiated by a node if it runs out of either type of task, and tasks are stolen from the corresponding end of the victim node. The double-ended version (rightmost bar, colored pink) does not initiate work stealing until the total number of tasks in the local queue drops below some threshold; when work stealing happens, half the stolen tasks come from each end of the victim's task queue. We find that the double-ended stealing strategy is consistently better, primarily because of the lower overheads of work stealing. Another variant of the single-ended stealing strategy with a single queue was also evaluated (colored blue, second bar from the right), where work stealing is initiated only when the GPU tasks in the queue fall below a threshold. This strategy consistently produces better results than does the basic single-ended stealing strategy. The reason is that the tasks at the GPU end are heavier, reducing the overhead of work stealing.

The second graph 8(b) shows experimental results for a task mix with twice the work in GPU-favored tasks as in CPU-favored tasks. Again, different skews in the initial distribution were evaluated. The third graph 8(c) shows results for a task mix that has twice as much work in CPU-favored tasks as GPU-favored tasks. The overall trend for these task mixes also shows that the double-ended queue strategy (rightmost bars, colored pink) and the selective single-ended GPU-steal strategy (colored blue, second from right) are consistently the superior schemes. In the next subsection, we present experimental results using these two work-stealing strategies for the CCSD(T) application.

Figure 9 shows a comparison of the execution time when different degrees of pipelining are used for different size input matrices. The x-axis is labeled (G, P), where G indicates the pipeline levels, or number of tasks in-flight, in the GPU and P indicates the number of pipelines in use for PGAS data movement. The y-axis shows the execution of executing the same task 1000 times. For the first two boxes, we have P set as *. This indicates that there is no communication overhead, and the data is already available in local memory. This represents the best time possible for pipelining of PGAS communication because there is no global address space communication.

The $(1, *)$ configuration indicates that the GPU tasks are issued sequentially and $(2, *)$ indicates that two streams are used for scheduling instructions in GPU. For the 1000×1000 matrix, we can observe the time as we increase the depth both in PGAS and GPU. Here, the total time for (2,2), (2,4) and (2,6) is almost same as (2,*). This indicates that the PGAS communication cost can be completely hidden from overlap achieved through pipelining.

The (4,8), (6,8), and (8,8) configurations also perform similarly. On the other hand, for 10×10 matrices, we observe a different result. This task has lower computation time than communication time. It can be easily observed that double buffering in both the GPU and CPU cannot hide the communication completely. After we reach level eight, the time to finish computation saturates. The result is also similar for 100×100 and 500×500 matrix sizes. From the graphs, it is clear that a greater degree of pipelining is needed for smaller tasks. We selected eight pipeline levels for small tasks based on these results.

DOI: 10.1002/cpe

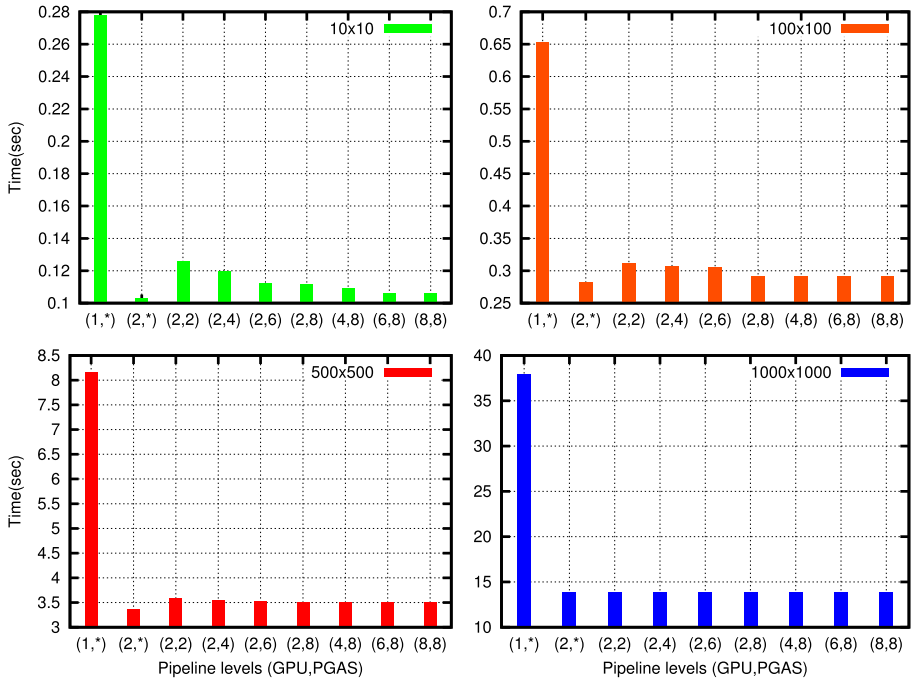


Figure 9. Performance impact of graphics processing unit (GPU) task execution and Partitioned Global Address Space (PGAS) communication pipelines for the matrix–matrix multiplication benchmark. Each experimental configuration (G, P) utilizes G GPU and P PGAS pipelines.

4.3. CCSD(T) application

We evaluated the work-stealing schemes using the CCSD(T) computation with four spatial blocks for the occupied orbitals ([16,5,16,5]) and virtual orbitals ([40,11,40,11]). The computation involves a sequence of block-sparse tensor contractions of different dimensionality, resulting in task mixes with significant variability across the tasks in terms of number of arithmetic operations. As with the matrix–matrix multiplication microbenchmark, the tasks with a smaller operation count are faster to execute on the CPU, whereas the heavier tasks are faster on the GPU. The distribution of tasks as a function of the execution time ratio on CPU versus GPU is shown in Figure 10. A ratio less than one indicates that the task is faster if executed on the host rather than on the attached GPU. One can see that a majority of the tasks perform better on the GPU. However, not all of these tasks are available at the same time. Each tensor contraction equation has a different set of tasks, and there are dependences between the equations. The overall execution thus involves a sequence of phases, with a set of independent contraction tasks to be executed in each phase and a barrier synchronization between the phases.

In the previous subsection, we presented experimental results assessing different queuing strategies under different initial distribution scenarios for the tasks. We report performance only for the best two work-stealing strategies: double-ended queue and single-ended-GPU-steal. Because the initial distribution of tasks in the queues does have an impact on performance, we consider several alternatives for the initial placement.

Affinity-based distribution. Locality-aware load balancing can help by hiding the communication overhead of accessing data from remote memory. In Scioto-ACC, each task provides an input interface and output interface. The data space is distributed over the nodes by using GA, which provides an interface to query where a particular block of data is physically located.

For collocation of tasks with their data, each task is initially placed in the queue of a process at the node where its output data is located. It is in general infeasible to ensure that both the input operands and the output are located on the same physical node; we prioritize data locality for the output because its update requires locking and synchronization.

DOI: 10.1002/cpe

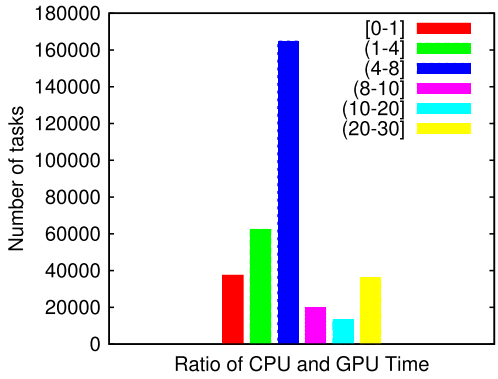


Figure 10. Histogram showing the distribution of the ratio of execution times for each time on the CPU over that on the graphics processing unit (GPU).

Heuristics-based distribution. Multiprocessor scheduling is an NP-hard problem. An often-used heuristic is to sort tasks based on their execution time and to assign tasks in decreasing order of execution time to the processor with the minimal total load so far. This is a $(4/3-1/3m)$ approximation algorithm, where m is the number of processors [47].

For Scioto-ACC, we need to schedule tasks on a heterogeneous system. Tasks are sorted based on the ratio of CPU time and GPU time and assigned by using the above heuristic, where each task is assigned to a node based on estimated finish time. If the ratio is more than one, the estimated time on GPU is considered. On the other hand, if the ratio is less than or equal to one, we distribute using the CPU finish time.

Optimal distribution based on execution time. We also generated an optimally load-balanced distribution of the tasks among the nodes by implementing a 0/1 integer program for the task mix. Although this is expensive and does not represent a practical online mapping strategy, it is useful in assessing how close to this we can get using heuristics. We note that while this optimal mapping finds the most load-balanced possible initial mapping, it does not guarantee the best finish time, because of inaccuracies in task execution time prediction and the fact that the overheads of work stealing are not modeled. However, we can expect to get a good initial task distribution. The objective function and the constraints for the integer linear programming are given below.

Following is the objective function of the formulation:

$$\min\left[\max_k\left(\sum_i W_{ik}^1 G_i, \sum_i W_{ik}^2 C_i\right)\right] \tag{1}$$

using constraints

$$\sum_k W_{ik}^1 + \sum_k W_{ik}^2 = 1. \tag{2}$$

For each task i, 0/1 variables W_{ik}^1 and W_{ik}^2 respectively denote whether the task is assigned to execute on the CPU of node k or the GPU of node k. The constraint ensures that each task is assigned a unique execution site. The objective function to be minimized is the maximum over all nodes, of the maximum accumulated time for the CPU and the GPU at that node. The assumption of this model is that each node has one CPU with an additional GPU.

Figure 11 shows the speedup of different strategies over the runtime of CPU after running $CCSD(T)$. We have used the single-queue, GPU-ended steal algorithm for this experiment. The CPU version executes on all cores.

It uses multithreaded DGEMM for the tensor contraction. We increased the number of nodes from 1 to 32. First, the red bar shows the timing for CPU all cores. But here we have used N processes for N core. The GPU-only bar shows the same tensor contraction code converted to run only on the GPU. In this case, the runtime for the CPU and GPU is almost, but not fully, equal. For one

DOI: 10.1002/cpe

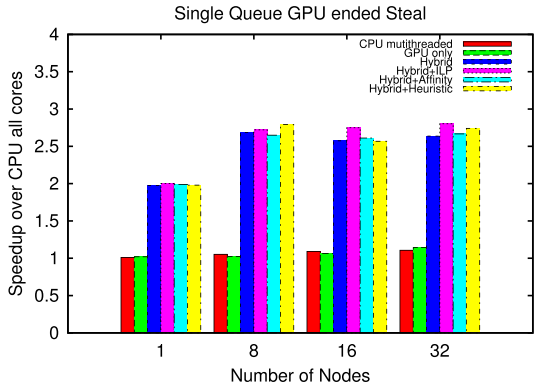


Figure 11. Execution time of CCSD(T) on Scioto-ACC using single-queue, graphics processing unit (GPU)-ended steal.

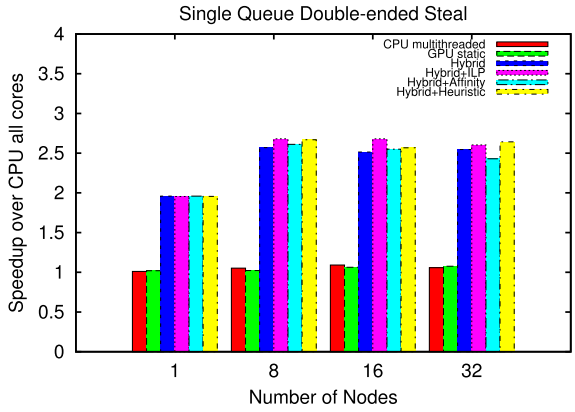


Figure 12. Execution time of CCSD(T) on Scioto-ACC using single-Queue, double-ended steal.

node, the other four schemes provide around 2× speedup. As task distribution has no effect, the total time for all these schemes is similar. For multiple nodes, however, we can see different results for different distributions. In the hybrid scheme, tasks are block distributed, and we obtained less than 2.5× speedup. For eight nodes, the Integer Linear Programming-based task distribution provides the best result. But for 32 nodes, we can see that the heuristic based on the task execution time provides the best performance. This can be explained by the work-stealing algorithm. Work stealing migrates tasks to different memory domains when a process is idle. Then, the initial distribution is no longer valid for some processes.

Figure 12 has the timing for $CCSD(T)$. But this uses a single-queue, double-ended steal. The trend of this graph matches our microbenchmark results. In some cases the single-queue, GPU-ended steal shows comparable performance with the single-queue, double-ended steal algorithm. However, the single-queue, GPU-ended steal is slightly better in most of the cases.

5. DISCUSSION AND FUTURE WORK

In this work, we presented the first step toward building a task parallel system that effectively utilizes both CPU and GPU resources. While we have tackled the immediate problems of constructing such a system, many additional challenges and opportunities abound.

5.1. Optimizing task queue structures

In the original Scioto implementation [11], the task queue is split into local and shared portions in order to reduce synchronization overheads incurred by the local process. However, in that work,

DOI: 10.1002/cpe

Scioto did not discriminate between tasks. In this work, we differentiate tasks based on an estimate of their execution time on the CPU and the GPU. A split queue can only support steal operations from the tail of the shared portion, which limits the type of tasks that can be stolen. Using a single, unified queue that supports stealing at both ends incurs higher locking overheads, but allows a thief to steal both CPU and GPU tasks, based on the availability of their resources.

Alternative approaches to balancing local overhead, communication cost, and load balance are a good topic for future work. For example, separate split queues can be used for storing CPU and GPU tasks. This reduces local synchronization overheads but increases the communication cost of steal operations. The relative availability of CPU and GPU resources is an additional factor to be considered. We evaluated our approach on a system with 32 nodes, each containing one GPU. For systems where the ratio of CPUs to GPUs differs, extensions and alternative approaches may be needed.

5.2. To steal or not to steal

In this work, we assume that any GPU task can be executed on the CPU, albeit with some opportunity cost in terms of lost performance benefits. We show the impact of executing matrix–matrix multiplication tasks, a key kernel in many applications, of different sizes in Figure 7. For this workload, the CPU provides better performance for small matrix dimensions, while the GPU provides significant speedups for large matrices. Given the significant performance gap between GPU and CPU for large matrices, attempting to steal additional CPU tasks may be a better use of CPU resources. The availability of such tasks is workload dependent, and additional mechanisms for detecting the characteristics of the workload and to allow users to provide hints may play a role in optimizing heuristics for when to steal and when to execute GPU tasks on the CPU.

5.3. Task execution cost modeling

We utilize a simple cost model that classifies tasks based on their anticipated execution time on the CPU versus on the GPU. While it is simple, our model is effective because it relies on relative, rather than absolute, performance characterization between the CPU and GPU and utilizes work stealing to resolve load imbalance. However, additional work to investigate more detailed cost models is warranted, in order to include factors such as the cost to move data versus the cost to move a task.

While classifying matrix–matrix multiplication tasks is relatively easy (based on the dimensions of the input matrices), this classification is more complex for many applications and may result in imprecise execution time estimates. In this work, we rely on dynamic load balancing to resolve inaccuracies in the cost model. However, additional approaches to cost modeling might be needed to tackle the broad space of application behaviors.

5.4. Management of CPU and graphics processing unit resources

Modern CPU and GPU processors contain many cores, and managing the assignment of cores to tasks is a complex problem. The optimal task scheduling problem is known to be NP-complete. We have therefore chosen a simpler model that is effective for a broad range of applications, while incurring low task scheduling overheads. In this model, all CPU tasks are assigned a fixed number of cores, which can be selected at the start of each parallel region, and a fixed number of tasks are dispatched to the GPU at any given time. Parallel efficiency can be improved by allowing individual tasks to specify varying resource requirements. Utilizing such information to optimize resource utilization is a challenging problem that needs further exploration.

6. CONCLUSIONS

We presented a compact approach for using accelerators in a task-parallel programming model context. Scioto provides a simple way to perform task-parallel programming. We extended this model for heterogeneous systems. A new work-stealing mechanism was proposed in which the queue

DOI: 10.1002/cpe

can contain tasks that can execute on either the CPU or GPU. Many variations of work-stealing algorithms were implemented in the context of GPU-based systems. We also implemented a pipelining mechanism for the GPU and partitioned global address space. Three strategies are presented to handle task distribution in the system. Blocked distribution and affinity-based distribution are evaluated in our library. A heuristic was proposed and evaluated for task scheduling in distributed heterogeneous systems.

We evaluated the Scioto-ACC library using a massively parallel tensor contraction kernel that demonstrates more than 2.5× speedup compared with a CPU kernel that uses all cores. In future work, we will automate the distribution of tasks. We also want to provide a generalized solution by using OpenCL.

REFERENCES

1. Esmaeilzadeh H, Blem E, Amant RS, Sankaralingam K, Burger D. Dark silicon and the end of multicore scaling. *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, IEEE, San Jose, CA, USA, 2011; 365–376.
2. Borkar S. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE* 2005; **25**(6):10–16.
3. Hoefler T, Schneider T, Lumsdaine A. Characterizing the influence of system noise on large-scale applications by simulation. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*: IEEE Computer Society, New Orleans, LA, USA, 2010; 1–11.
4. UPC C. UPC language specifications, v1.2. *Technical Report LBNL-59208*, Lawrence Berkeley National Lab, 2005.
5. Mellor-Crummey J, Adhianto L, Scherer WN III, Jin G. A new vision for coarray fortran. *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, ACM, New York, NY, USA, 2009; 5:1–5:9.
6. Numrich RW, Reid J. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum* 1998; **17**(2):1–31.
7. Yelick KA, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger PN, Graham SL, Gay D, Colella P, Aiken A. Titanium: a high-performance java dialect. *Concurrency - Practice and Experience* 1998; **10**(11-13): 825–836.
8. Dinan J. Scalable task parallel programming in the partitioned global address space. *Ph.D. Thesis*, The Ohio State University, 2010.
9. Min SJ, Iancu C, Yelick K. Hierarchical work stealing on manycore clusters. *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, Galveston Island, Texas, USA, 2011.
10. Jose J, Potluri S, Luo M, Sur S, Panda D. Upc queues for scalable graph traversals: design and evaluation on infiniband clusters. *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, Galveston Island, Texas, USA, 2011.
11. Dinan J, Krishnamoorthy S, Larkins DB, Nieplocha J, Sadayappan P. Scioto: a framework for global-view task parallelism. *Proceedings of the 2008 37Th International Conference on Parallel Processing*, ICPP '08, Portland, Oregon, USA, 2008; 586–593.
12. Sidelnik A, Maleki S, Chamberlain BL, Garzar'n MJ, Padua D. Performance portability with the chapel language. *Parallel and Distributed Processing Symposium, International* 2012; **0**:582–594.
13. Nieplocha J, Harrison RJ, Littlefield RJ. Global arrays: a portable "shared-memory" programming model for distributed memory computers. *Supercomputing (sc)*, Washington, DC, USA, 1994; 340–349.
14. Nieplocha J, Palmer B, Tipparaju V, Krishnan M, Trease H, Aprà E. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications* 2006; **20**(2):203–231.
15. Kendall RA, Aprà E, Bernholdt DE, Bylaska EJ, Dupuis M, Fann GI, Harrison RJ, Ju J, Nichols JA, Nieplocha J, Straatsma TP, Windus TL, Wong AT. High performance computational chemistry: an overview of NWChem a distributed parallel application. *Computer Physics Communications* 2000; **128**(1-2):260 –283.
16. Amos RD, Bernhardsson A, Berning A, Celani P, Cooper DL, Deegan MJO, Dobbyn AJ, Eckert F, Hampel C, Hetzer G, Knowles PJ, Korona T, Lindh R, Lloyd AW, McNicholas SJ, Manby FR, Meyer W, Mura ME, Nicklass A, Palmieri P, Pitzer R, Rauhut G, Schütz M, Schumann U, Stoll H, Stone AJ, Tarroni R, Thorsteinsson T, Werner HJ. MOLPRO, a package of ab initio programs designed by H. *J. Werner and PJ Knowles, version* 2002; **2002**.
17. Oehmen C, Nieplocha J. Scalablast: a scalable implementation of blast for high-performance data-intensive bioinformatics analysis. *IEEE Transactions on Parallel and Distributed Systems* 2006:740–749.
18. Blumofe R, Leiserson C. Scheduling multithreaded computations by work stealing. *Proceedings of the 35Th Symposium on Foundations of Computer Science (FOCS)*, Santa Fe, New Mexico, USA, 1994; 356–368.
19. Cong G, Kodali S, Krishnamoorty S, Lea D, Saraswat V, Wen T. Solving irregular graph problems using adaptive work-stealing. *Proceedings 2008 International Conference on Parallel Processing (37Th ICPP'08) CD-ROM*, Portland, OR, 2008; 536–545.
20. Frigo M, Leiserson CE, Randall KH. The implementation of the Cilk-5 multithreaded language. *PLDI*, Montreal, QC, Canada, 1998; 212–223.
21. Kumar V, Grama AY, Vempaty NR. Scalable load balancing techniques for parallel computers. *J. Parallel Distrib. Comput.* 1994; **22**(1):60–79.

DOI: 10.1002/cpe

22. Tzeng S, Patney A, Owens JD. Task management for irregular-parallel workloads on the gpu. *Proceedings of the Conference on High Performance Graphics*, HPG '10, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2010; 29–37.
23. Ravi VT, Ma W, Chiu D, Agrawal G. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. *Proceedings of the 24Th ACM International Conference on Supercomputing*, ICS '10, ACM, New York, NY, USA, 2010; 137–146.
24. Chatterjee S, Grossman M, Sbrlea A, Sarkar V. Dynamic task parallelism with a gpu work-stealing runtime system. In *Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, Vol. 7146, Rajopadhye S, Mills Strout M (eds). Springer Berlin Heidelberg, 2013.
25. Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioglu K, von Praun C, Sarkar V. X10: an object-oriented approach to non-uniform cluster computing. *International Conference Object-Oriented Programming, Systems, Languages, And Applications (OOPSLA)*, ACM SIGPLAN, 2005; 519–538.
26. Cunningham D, Bordawekar R, Saraswat V. Gpu programming in a high level language: compiling x10 to cuda. *Proceedings Of The 2011 ACM SIGPLAN X10 Workshop*, X10 '11, ACM, New York, NY, USA, 2011; 8:1–8:10.
27. Chamberlain BL, Callahan D, Zima HP. Parallel programmability and the chapel language. *Intl. J. High Performance Computing Applications (IJHPCA)* 2007; **21**(3):291–312.
28. Sidelnik A, Chamberlain BL, Garzaran MJ, Padua D. Using the high productivity language chapel to target gpgpu architectures 2011. Tech report. (Available from: https://www.ideals.illinois.edu/handle/2142/18874).
29. Saraswat VA, Kambadur P, Kodali S, Grove D, Krishnamoorthy S. Lifeline-based global load balancing. *SIGPLAN Not.* February 2011; **46**(8):201–212.
30. Strengert M, Müller C, Dachsbacher C, Ertl T. CUDASA: Compute unified device and systems architecture. In *EGPGV*, Favre JM, Ma KL (eds). Eurographics Association, 2008; 49–56.
31. NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, Vol. PG-02829-001_v5.0. NVIDIA Corporation, 2012.
32. Xiao S, Balaji P, Dinan J, Zhu Q, Thakur R, Coghlan S, Lin H, Wen G, Hong J, Feng WC. Transparent accelerator migration in a virtualized GPU environment. *Proceedings 12Th IEEE/ACM Intl. Conf. On Cluster, Cloud, And Grid Computing*, CCGrid '12, Ottawa, Canada, 2012; 124–131.
33. Bueno J, Martinell L, Duran A, Farreras M, Martorell X, Badia RM, Ayguade E, Labarta J. Productive cluster programming with ompss. *Proceedings of the 17Th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, Springer-Verlag; Berlin, Heidelberg, 2011; 555–566.
34. Augonnet C, Thibault S, Namyst R, Wacrenier PA. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 2011; **23**:187–198.
35. Becchi M, Byna S, Cadambi S, Chakradhar S. Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. *Proceedings of the 22Nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, ACM, New York, NY, USA, 2010; 82–91.
36. Grewe D, OBoyle M. A static task partitioning approach for heterogeneous systems using opencl. *Compiler construction*: Springer, Saarbrücken, Germany, 2011; 286–305.
37. Bosilca G, Bouteiller A, Herault T, Lemarinier P, Saengpatsa NO, Tomov S, Dongarra JJ. Performance portability of a gpu enabled factorization with the dague framework. *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*: IEEE, Austin, Texas, USA, 2011; 395–402.
38. Frey S, Ertl T. Patraco: a framework enabling the transparent and efficient programming of heterogeneous compute networks. *Proceedings of the 10Th Eurographics Conference on Parallel Graphics and Visualization*, EG PGV'10, Eurographics Association, Norrköping, Sweden, 2010; 131–140.
39. Tomov S, Dongarra J, Baboulin M. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Computing* 2010; **36**(56):232 –240.
40. Huo X, Ravi VT, Agrawal G. Porting irregular reductions on heterogeneous cpu-gpu configurations. *High Performance Computing (HIPC), 2011 18Th International Conference on*, IEEE, Bangalore, India, 2011; 1–10.
41. Shirahata K, Sato H, Matsuoka S. Hybrid map task scheduling for gpu-based heterogeneous clusters. *Cloud Computing Technology And Science (CLOUDCOM), 2010 IEEE Second International Conference On*, IEEE, Indianapolis, IN, USA, 2010; 733–740.
42. Diamos GF, Yalamanchili S. Harmony: an execution model and runtime for heterogeneous many core systems. *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, ACM, New York, NY, USA, 2008; 197–200.
43. Dinan J, Larkins DB, Sadayappan P, Krishnamoorthy S, Nieplocha J. Scalable work stealing. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, Portland, Oregon, USA, 2009; 1–11.
44. Intel. *Intel Compilers 12 for Linux*. http://softwares.intel.com [Accessed on 2012].
45. Bartlett RJ, Musiał M. Coupled-cluster theory in quantum chemistry. *Reviews of Modern Physics* 2007; **79**:291–352.
46. Baumgartner G, Auer A, Bernholdt DE, Bibireata A, Choppella V, Cociorva D, Gao X, Harrison RJ, Hirata S, Krishanmoorthy S, Krishnan S, Lam C, Qingda L, Nooijen M, Pitzer RM, Ramanujam J, Sadayappan P, Sibiryakov A. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE* 2005; **93**(2):276–292. special issue on "Program Generation, Optimization, and Adaptation".
47. Graham RL. Bounds on multiprocessing timing anomalies. *Siam Journal on Applied Mathematics* 1969; **17**(2): 416–429.

DOI: 10.1002/cpe