HabaneroUPC++: a Compiler-free PGAS Library*

Vivek Kumar†, Yili Zheng‡, Vincent Cavé†, Zoran Budimlić†, and Vivek Sarkar†

†Rice University

‡Lawrence Berkeley National Laboratory

The Partitioned Global Address Space (PGAS) programming models combine shared and distributed memory features, providing the basis for high performance and high productivity parallel programming environments. UPC++ [39] is a very recent PGAS implementation that takes a library-based approach and avoids the complexities associated with compiler transformations. However, this implementation does not support dynamic task parallelism and only relies on other threading models (e.g., OpenMP or pthreads) for exploiting parallelism within a PGAS `place`.

In this paper, we introduce a compiler-free PGAS library called HabaneroUPC++, which supports a tighter integration of intra-`place` and inter-`place` parallelism than standard hybrid programming approaches. The library makes heavy use of C++11 lambda functions in its APIs. C++11 lambdas avoid the need for compiler support while still retaining the syntactic convenience of language-based approaches. The HabaneroUPC++ library implementation is based on a tight integration of the UPC++ library and the Habanero-C++ library, with new extensions to support the integration. The UPC++ library is used to provide PGAS communication and function shipping support using GASNet, and the Habanero-C++ library is used to provide support for intra-`place` work-stealing integrated with function shipping. We demonstrate the programmability and performance of our implementation using two benchmarks, scaled up to 6K cores. The insights developed in this paper promise to further enhance the usability and popularity of PGAS programming models.

Categories and Subject Descriptors

D1.3 [**Programming Techniques**]: Concurrent Programming – distributed programming; parallel programming; D3.3 [**Programming Languages**]: Language Constructs and Features – concurrent programming structures; D3.4 [**Programming Languages**]: Processors – compilers; optimization; run-time environments

*This work is supported by the X-Stack program funded by the U.S. Department of Energy Office of Advanced Scientific Computing Research. Any opinions, findings and conclusions expressed herein are the author's and do not necessarily reflect those of the sponsors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

General Terms

Design, performance

Keywords

PGAS, UPC++, Habanero, scheduling, work-stealing

1. INTRODUCTION

The *Partitioned Global Address Space* (PGAS) programming model [2] strikes a balance between shared and distributed memory models [26]. It provides ease of programming due to its global address memory model and performance due to locality awareness. Languages belonging to this category include Co-Array Fortran [23], Titanium [37], UPC [10], X10 [6] and Chapel [5]. They rely on compiler transformations to convert the user code to the native code. Some of these languages, such as Titanium, X10 and Chapel, use code transformations to provide dynamic tasking capabilities using a work-stealing scheduler [22, 12, 18, 19, 27, 4] for load balancing of the dynamic tasks. Min et al. introduced API based dynamic tasking library for UPC [21]. However, it lacks the expressiveness of the X10's `finish`-async style dynamic tasking. Co-Array Fortran does not allow dynamic tasking but permits the user to use OpenMP libraries for achieving loop-level parallelism.

The library approach to PGAS programming makes it easier to interoperate with other programming models such as MPI [7], CUDA [24] or even other PGAS languages. It also avoids the significant development and maintenance costs associated with a language-based approach. In addition, the compiler-free approach facilitates adding new features and combining multiple packages. UPC++ [39] is a very recent PGAS approach, implemented as a C++ library. However, UPC++ does not allow intra-`place` [1] dynamic tasking. It introduces inter-`place` asynchronous copy and asynchronous function shipping, with the limitation that these asynchronous activities don't automatically migrate (e.g., no work-stealing) and `finish` doesn't track transitively spawned descendant activities by default as in X10's `finish-async`. For intra-`place` loop parallelism in UPC++, the user relies on other threading models (e.g., OpenMP or pthreads).

Work-stealing schedulers have emerged as the approach of choice for dynamic load balancing. Within the underlying language runtime they use a fixed size worker pool to schedule the work exposed by the programmer, exploiting idle processors and unburdening those that are overloaded. Habanero-Java [4] and Habanero-C [28] implement X10 style `finish-async` tasks for shared memory. A variant of Habanero-C called HCMPI [7] provides SPMD programming model using intra-node `finish-async` parallelism and MPI for inter-node parallelism. However, HCMPI relies on

[1] We use *place* to represent each independent execution unit in UPC++ and *threads* to represent work-stealing workers.

```
finish {
  async [(place)] [IN (var1, var2, ...)]
                  [OUT (var1, var2, ...)]
                  [INOUT (var1, var2, ...)]
                  [AWAIT (ddf1, ddf2, ...)]
                  [phased] Statement;

  forasync        [in (var1, var2, ...)]
                  [point (ind1, ind2, ...)]
                  [size (siz1, siz2, ...)]
                  [seq (seq1, seq2, ...)] Body;
}
```

Figure 1: Dynamic tasking constructs in Habanero-C.

compiler transformations and by using MPI it does not get the benefits of the PGAS programming model.

The main contributions of this paper are: a) Habanero-C++ library, a compiler-free approach for using Habanero work-stealing using C++11 lambda functions; b) HabaneroUPC++, a new PGAS library combining the benefits of Habanero-C++ and UPC++, and allowing the programmer to conveniently and concisely expose dynamic task parallelism in a highly scalable PGAS implementation; and c) evaluation of HabaneroUPC++ using two benchmarks, scaled up to 6K cores.

The rest of the paper is structured as follows. Section 2 provides the relevant background. Section 3 summarizes the HabaneroUPC++ programming model. Section 4 explains the details of our runtime. Section 5 evaluates HabaneroUPC++ performance on the Edison supercomputer at NERSC. Section 6 discusses the related work and, section 7 concludes the paper.

2. BACKGROUND

2.1 Habanero-C

Habanero-C language provides a `finish`-`async` task-parallel programming model [4] for exploiting intra-node parallelism. Here we briefly describe some of its features related to this paper, more details can be found in [28]. Habanero-C is based on Habanero-Java, which itself was derived from an earlier version of X10. Figure 1 shows the different dynamic tasking constructs available in Habanero-C.

The `async` is used to create a child task asynchronously to execute `Statement`. `finish` is used to join all `async` tasks, including the transitively spawned ones. Habanero-C uses compiler transformations to translate these dynamic tasking constructs from user code to runtime calls. The local variables declared outside of the `async` scope can be acessed inside of the `async` by using `IN/OUT/INOUT` clauses. `IN` declares a variable as read-only, `OUT` specifies write-only and `INOUT` specifies both read and write access. The `place` can be used to specify a node within a hierarchical place tree [35] (intra-node only). The `AWAIT` clause can be used for data-driven task (DDT) synchronization [33]. `async` with `AWAIT` clause gets executed only when all the `ddf` (data-driven futures or DDFs) have been satisfied (written using a `put` operation). The `phased` clause registers an `async` with phasers [29] specified in the list with the corresponding modes. If empty phaser list is specified then it registers the `async` on all the phasers of the parent task. The phased `async` is executed only when the signal modes are satisfied on the listed phasers.

`forasync` construct is similar to a program loop which exhibits `parallel_for` parallelism. The `point` clause specifies the loop indices in each dimension. Total number of iteration in each dimension is specified using the `size` clause. The clause `seq` specifies the tile size. The runtime can be instructed to schedule `forasync` in two ways, *chunked scheduling* and *recursive scheduling*. In chunked mode the loop iterations are chunked into blocks of length specified by `seq` clause whereas in recursive mode the iterations are recursively partitioned until size matches `seq` value.

2.2 OCR

Studies suggest future high performance computing systems would scale to *exascale*. These exascale systems would contain attributes that would be 1000 times the value of similar attribute of a petascale system from the year 2010 [1]. These systems will be massively multicore per chip. Their performance will be driven by parallelism, constrained by energy and data movement. Open Community Runtime (OCR) [25] is an open-source, multi-institutional project aimed at creating a common set of runtime APIs for task-parallel programming models suited for the exascale systems. These APIs can be either targeted by a compiler for a high-level programming language, or called directly by a hero programmer writing an application directly in OCR.

OCR also supplies a reference implementation that uses the ideas from the Habanero-C (Section 2.1) work-stealing runtime to implement non-blocking load-balancing of tasks across different processors. As of this writing, there are OCR implementations on x86 shared-memory multiprocessors and on clusters of x86 machines.

The main concepts in OCR are:

1. Tasks. *Event-driven tasks* (EDTs) are the units of computation in OCR. All EDTs have to declare a set of *dependencies* to which external events can be connected. An EDT does not begin execution until all its dependencies have been satisfied. EDTs are intended to be functional, non-blocking pieces of code. Internally, they can exploit some other form of parallelism (such as data parallelism) but they should communicate with other EDTs only by using Data Blocks and Events defined below. All EDTs have a globally unique ID (GUID) that identifies them across the system.

2. Data. *Data Blocks* (DBs) are the mechanism for storing and communicating data between EDTs. DBs are identified by a GUID as well. Since DBs can be relocated and replicated by the runtime for performance, energy or resilience reasons, it is essential that all user data is expressed in terms of DB GUIDS and offsets within DBs instead of pointers as in traditional shared-memory systems.

3. Events. *Events* are the mechanism for creating data and control dependencies in OCR. When an EDT's dependence is connected to an event, the runtime is informed that that particular EDT depends on that particular event, and the EDT will not execute until that event is satisfied. An EDT performing a put on an event can satisfy that event. A put on an event can be empty (used for triggering the execution of EDTs waiting on that event, i.e. implementing control dependence) or it can take DB GUID as a parameter (used to pass data to the EDTs waiting on that event, i.e. implementing data dependence). Each event has a GUID as well.

The programmer (or compiler) creates an OCR program by constructing a dynamic graph of EDTs, DBs and events. There are currently several higher-level parallel programming models that map onto OCR, including CnC [14], HClib [3], HTA [11] and RStream [20]. This paper demonstrates the integration of OCR (through the use of the HCLib Habanero implementation built on top of OCR) with the UPC PGAS programming model.

```
// struct packing all vars
typedef struct {
  var1;
  var2;
  ....
} async_args_t;

// Wrapper function to execute user function
void async_func(void* async_args) {
  // Cast pointers
  async_args_t *args
          = (async_args_t *) async_args;
  // Pass all local vars to user function
  Statement(args->var1, args->var2, ....);
}

main() {
  // Pack all ddf in a list
  ddf_list = create_ddf_list(ddf1, ddf2, ....);
  // Pack all phaser in a list
  phaser_list = create_phaser_list(....);
  // Pack all local vars in a struct
  async_args_t async_args = {var1, var2, ....};
  // Start finish scope
  start_finish();
  // Schedule the async task
  schedule_async(Statement, &async_args, place,
                      ddf_list, phaser_list);
  // End finish scope
  end_finish();
}
```

Figure 2: Using HClib's approach to implement Habanero-C's `async` task in Figure 1.

2.3 HClib

OCR is implemented as a runtime that can be targeted by higher-level programming systems. HClib is a C-language based library, which is implemented on top of OCR to provide the dynamic tasking features of Habanero-C language (Section 2.1). However, this approach comes at the cost of losing Habanero-C language's syntactic constructs and its compiler support for continuations. Figure 2 shows the pseudo code to implement the Habanero-C `async` from Figure 1 using HClib. The `Statement` needs to be wrapped inside a function, which the OCR runtime can execute. All the local variables from outer scope, which are to be accessed inside `Statement`, have to be wrapped in a `struct` and passed as a "`void*`" pointer to OCR. Similarly, all the DDFs and phasers are also packed in a list.

The HClib library is built on top of OCR. It is composed of three layers: HClib; HClib on OCR; and OCR. The first layer defines HClib data-structures and user APIs. It is agnostic to the underlying runtime it executes on. The second layer acts as a bridge between HClib and OCR. This layer knows about the OCR runtime and translates HClib actions into OCR ones. The third layer is the OCR runtime, which is the supporting execution platform. Most of the interaction between HClib and OCR happens in the second layer. In this layer, the API call for any of the Habanero's asynchronous task types is transformed into a generic EDT representation. HClib relies on OCR to configure and bootstrap the underlying execution runtime.

2.4 UPC++

UPC++ is a PGAS C++ extension, which can be used as a standalone programming system for developing PGAS C++ applications or as a runtime component to support other high-level programming languages and libraries. UPC++ adopts the PGAS memory model and the SPMD execution model from UPC. Each UPC++ `place` has its private address space and a partition of the global address space, in which data is directly accessible by all UPC++ places even on distributed-memory systems. Here we give a brief overview of UPC++ features used in HabaneroUPC++ about global data sharing, one-sided communication, and remote function invocation (a.k.a. function shipping). Table 1 lists the essential UPC++ programming constructs.

Table 1: Basic PGAS primitives in UPC++

Programming Idiom	UPC++
Number of places	`ranks()`
My ID	`myrank()`
Shared variable	`shared_var<Type> v`
Shared array	`shared_array<Type>` a(count)
Global pointer	`global_ptr<Type> p`
Memory allocation	`allocate<Type>(place, count)`
Data transfer	`async_copy<Type>(src, dst, count)`
RPC	`async(place)(Function, args...)`
Synchronization	`async_wait()/async_try()/barrier()`

In the UPC++ PGAS memory model, shared objects can be declared statically at compile time or allocated dynamically at run time. UPC++ shared data types are implemented as generic templates parameterized over the object type, which can be either built-in types or user-defined types (e.g., `class`. `shared_var<T>` type of data are physically located in the same global partition as in UPC, and `shared_array<T>` type of data are block-cyclically distributed across all global partitions. UPC++ `shared_array` can be initialized with dynamic size and blocking factor at runtime (e.g. `sa.init(N, BF)`). The default blocking factor of UPC++ shared arrays is 1 (cyclic distribution) and it can be changed with `shared_array` member function `set_blk_sz`. The subscript operator `[]` is overloaded to provide the same accessing rules as non-shared arrays. In addition, `shared_array` can be declared (collectively) inside a function scope. Due to its SPMD nature, UPC++ shared variable names can be referenced from different processes, which provides a convenient way for communication and synchronization. Regardless of their physical location, shared objects are accessible by any UPC++ `place`.

Dynamic global memory allocation is done through generic global pointer type (`global_ptr<T>`) which points to a shared object of type `T`. A global pointer encapsulates both the process `place` and the local address of the shared object referenced by the pointer. Pointer arithmetic with global pointers in UPC++ works the same way as arithmetic on regular C++ pointers. Global address space memory can be allocated and freed at any `place` by `allocate` and `deallocate` templated functions.

Communication in UPC++ applications may appear in two forms: 1) explicit data transfer using one-sided `copy` functions; 2) implicit data communication when shared objects appear in an expression. For example, if a shared object is on the left-hand-side of an assignment statement then it's equivalent to a `put` operation. Likewise, it is a `get` operation if the shared object is on the right-hand-side. In UPC++, the type conversion operator to the local object type (`operator T()`) is overloaded for accessing remote shared objects. The user can initiate bulk data movement operations using the `copy` function or its non-blocking counterpart `async_copy` function, for which the `src` and `dst` buffers are assumed to be contiguous. `async_copy` enables overlapping communication with computation or other communication. The completion status of `async_copy` can be queried by `async_try` or

```
finish ( [capture_list]() {
  async ( [capture_list]() {
    S1;
  }); // end async
  asyncAwait(ddf1, ..., [capture_list]() {
    S2;
  }); // end asyncAwait
  asyncPhased(ph1, mode1,..., [capture_list]() {
    S3;
  }); // end asyncPhased
  forasync(dim, style, ind1, ..., siz1, ...,
           seq1, ..., [capture_list]() {
    S4;
  }); // end forasync
});   // end finish
```

Figure 3: Dynamic tasking using C++11 lambda functions.

waited by `async_wait`. Finally, user may register an `async_copy` operation with an `event` that can be synchronized later. UPC++ also provides MPI-style collective operations implemented on top of the GASNet collectives API.

An important new feature in UPC++ but not in UPC is remote function invocation. The user may start an asynchronous remote function with the `async` construct and specify dependencies among distributed tasks using the *event* mechanism similar to Phalanx [13]. Each `async` function call may be registered with an `event` that will be signaled after the remote function is completed, and used as a precondition to launch later `async` operations. UPC++ remote function invocation is implemented on top of GASNet active messages.

3. HabaneroUPC++ PROGRAMMING MODEL

In this section we first describe the Habanero-C++ dynamic tasking library. Being a C++ library, Habanero-C++ easily integrates with UPC++ and offers a highly scalable PGAS implementation, which we refer to as HabaneroUPC++. We also discuss the features of this new programming model.

3.1 Habanero-C++ Dynamic Tasking Library

Habanero-C++ uses C++11 lambda functions to express Habanero-C dynamic tasking constructs (Section 2.1). As we take a compiler-free approach, the program syntax slightly differs from Habanero-C. Figure 3 shows the Habanero-C++'s equivalent of the Habanero-C's dynamic tasking constructs shown in Figure 1. The syntax "`[capture_list]()`" marks the beginning of a C++11 lambda function. The capture-list contains the list of variables, which we want to use inside statements `S1`, `S2`, `S3` and `S4`. These variables can either be captured by reference or by value. Passing a single "`&`" as the capture-list captures all the local variables by reference while passing a "`=`" captures all the local variables by value. Asynchronous tasks do not return values. However, parameter-result variables can be passed as references in the capture-list. The capture-list of lambda functions provides a mechanism to pass variables to an async that is semantically equivalent to the usage of the `IN/OUT/INOUT` clauses of Habanero-C. Relying on C++11 features, Habanero-C++ currently provides support for `async`'s `AWAIT` and `PHASED` clauses. The implementation can be further extended to support accumulators [30] and places [35]. Similarly, Habanero-C++ provides support for the `forasync` construct (Figure 3). The `dim` argument to the `forasync` call specifies the loop dimension. The `style` argument specifies whether to use the chunked or recursive scheduling. For each dimension, the user provides its lower-

```
asyncCopy(global_ptr<T> src, global_ptr<T> dst,
          size_t count, DDF* ddf=NULL);
```

(a) Asynchronous copy with optional DDF

```
asyncAt(place, [capture_list]() {
  S1;
});
```

(b) Asynchronous remote lambda invocation

Figure 4: Remote asynchronous calls.

bound (`ind1`), size (`size1`) and tile size (`seq1`). If the tile sizes arguments are left as zero, the runtime assigns a default size, which is the total number of iterations (`size`) divided by the total number of work-stealing threads.

3.2 HabaneroUPC++ Programming Features

In UPC++ the execution units (`place`) are single threaded unless combined with OpenMP to achieve the loop level parallelism. This style of parallelism is restrictive and does not enjoy the benefits of work-stealing schedulers. Other kinds of parallelism can be effectively load-balanced using work-stealing and include divide and conquer, irregular graph computations and loop parallelism. HabaneroUPC++ integrates the benefit of Habanero-C++'s work-stealing library in UPC++. Similar to UPC++, the HabaneroUPC++ program also starts in a SPMD fashion, where each `place` gets a copy of the `main` function. Taking UPC++ as baseline, we will now discuss our newly added features.

3.2.1 Asynchronous Remote Copy

As discussed in Section 2.4, UPC++ provides a non-blocking copy function. To be able to integrate with Habanero-C++, we provide a variant of this as shown in Figure 4(a). The optional DDF allows launching an `asyncAwait`, which gets scheduled only when the `asyncCopy` is complete.

3.2.2 Asynchronous Remote Function Invocation

UPC++ provides its own version of `async` for remote function invocations. However, these remote `async` does not capture the closure of the `async` spawn. Moreover, threads within a UPC++ `place` cannot call them concurrently. HabaneroUPC++ provides a variant of UPC++'s `async`. This is called `asyncAt` and its syntax is shown in Figure 4(b). This `asyncAt` can be nested, they are thread-safe, and can also calls Habanero-C++'s `async` inside the lambda function.

3.2.3 Joining Asynchronous Tasks

To be able to join all the asynchronous tasks (`async`, `asyncAwait`, `asyncPhased`, `forasync`, `asyncAt` and `asyncCopy`), HabaneroUPC++ provides a special version of finish called as `finish_spmd`, shown in Figure 5. `finish_spmd` waits for all the dynamically spawned asynchronous tasks in its scope, which includes remote asynchronous tasks as well. HabaneroUPC++ also allows arbitrary nesting of `finish` inside `finish_spmd`. However, this `finish` only allows the launch of Habanero-C++ asynchronous tasks and no remote asynchronous tasks.

3.2.4 Collective Communications

For collective communications, HabaneroUPC++ relies on UPC++ collectives and does not modify their default implementation. However, HabaneroUPC++ restricts the usage of collectives

```
finish_spmd ([capture_list]() {
  async(...);         // local
  asyncAwait(...);    // local
  asyncPhased(...);   // local
  forasync(...);      // local
  asyncCopy(...);     // remote
  asyncAt(...);       // remote
});
```

Figure 5: Joining of asynchronous tasks using special finish.

```
void async_wrapper(void* args) {
  // Cast the lambda object
  std::function<void()> *lambda =
                (std::function<void()> *)args;
  // Execute the lambda
  (*lambda)();
  delete lambda;
}

void schedule_async(bool inter_place,
                 std::function<void()> lambda) {
  // Heap allocate a lambda object
  // as currently its on stack
  std::function<void()> * lambda_copy =
                new std::function<void()> (lambda);
  // bookkeeping: OCR increments async count
  // which decrements once this task has executed
  runtime(inter_place, async_wrapper, lambda_copy);
}

void async(std::function<void()> lambda)
{
  // Pass the lambda to work-stealing runtime
  // as an intra-place asynchronous task
  schedule_async(false, lambda);
}
```

Figure 6: Implementation of `async`

inside
`finish_spmd`. UPC++ collectives are discussed in Section 2.4.

4. IMPLEMENTATION

The previous section explains the HabaneroUPC++ programming model. In this section we describe the implementation of the HabaneroUPC++ runtime.

4.1 Translating C++11 Lambdas to Runtime Calls

HabaneroUPC++ takes a compiler-free approach. Hence, the API exposed to the user relies on a set of HabaneroUPC++ runtime calls that can handle C++11 user-defined lambdas. In a nutshell, the C++ compiler converts a lambda function into a `class` with an overloaded "`()`" operator, which acts as a function. The variables captured (e.g., `capture_list` in Figure 3) in the lambda definition become member variables of this `class`. This class also has access to all the global variables in the program scope. The variables can be captured either by value or by reference. In case an object is captured by value, then the copy constructor is called when the lambda closure is created. The code to create lambda closure is automatically generated by the C++ compiler. By default all variables captured by value are immutable, unless the `mutable` keyword is explicitly used. Passing a lambda as a function parameter is similar to passing a `class` object. A detailed explanation of the C++ implementation of lambdas is available in [16].

We now describe how the user lambda function gets communicated to the runtime. The runtime treats lambda functions dif-

```
// Executes at destination
template <typename T>
void asyncAt_wrapper(global_ptr<T> lambda) {
  // Allocate memory in myPlace's partition
  // in global address space
  upcxx::global_ptr<T> my_lambda =
                upcxx::allocate<T>(myPlace);
  // Copy the lambda in memory allocated above
  upcxx::copy(remote_lambda, my_lambda);
  // Execute the lambda
  (*(T*)my_lambda)();
  // Free memory allocated for lambdas
  deallocate(my_lambda);
  deallocate(remote_lambda);
  atomic(incoming_tasks++); // bookkeeping
}

template <typename T>
void asyncAt(int toPlace, T lambda) {
  if(toPlace == myPlace) {
    // Run as a local async
    async(lambda);
  }
  else {
    atomic(outgoing_tasks++); // bookkeeping
    // Allocate memory in myPlace's partition
    // in global address space
    upcxx::global_ptr<T> remote_lambda =
            upcxx::allocate<T>(myPlace);
    // Copy the lambda in memory allocated above
    memcpy(remote_lambda, &lambda, sizeof(T));
    // Create a lambda to execute UPC++ calls
    auto lambda = [=]() {
      // Use UPC++ active message to invoke
      // asyncAt_wrapper at toPlace
      upcxx::async(toPlace)(asyncAt_wrapper<T>,
                                remote_lambda);
    };
    // Schedule an inter-place async task
    schedule_async(true, lambda);
  }
}
```

Figure 7: Implementation of `asyncAt`

```
// Executes at source once asyncCopy is complete
void perform_ddfWrite(void* ddf) {
  if(ddf != NULL) DDF_PUT(ddf);
  atomic(incoming_tasks++); // bookkeeping
}

template <typename T>
void asyncCopy(global_ptr<T> src,
               global_ptr<T> dst,
               size_t count, DDF_t* ddf=NULL) {
  atomic(outgoing_tasks++); // bookkeeping
  // Create a lambda to execute UPC++ calls
  auto lambda = [=]() {
    // Use UPC++ asynchronous copy function
    // to perform remote copy and tie it with
    // a UPC++ event object
    upcxx::event e;
    upcxx::async_copy(src, dst, count, &e);
    // Attach a callback function which waits
    // on event e. Once async_copy is done, this
    // callback executes at current place and
    // launches function perform_ddfWrite with
    // ddf as parameter
    upcxx::async_after(myPlace, &e)(perform_ddfWrite,
                                      (void*)ddf);
  };
  // Schedule an inter-place async task
  schedule_async(true, lambda);
}
```

Figure 8: Implementation of `asyncCopy`

ferently depending on whether they represent asynchronous tasks to be executed locally (`async`, `asyncAwait`, `asyncPhased` and `forasync`) or remotely (`asyncAt` and `asyncCopy`). Figure 6 shows the implementation of the `async` construct. Because an `async` is potentially executed after its creation context is done executing, the runtime cannot rely on any variables that may have been stack allocated. For that reason, the runtime creates a heap-allocated copy of the user-defined lambda. Once the lambda is passed to the work-stealing runtime, the worker threads can execute it by calling the `async_wrapper` function and use the overloaded "`()`" operator to convert the lambda into a function call. The lambdas for `asyncAwait`, `asyncPhased` and `forasync` are treated in similar fashion, but they can also take a variable number of arguments. For instance, `asyncAwait` can get multiple DDFs; `asyncPhased` can have multiple phasers and signal modes; and `forasync` can be of any dimension. To that effect, the runtime provides several copies of `asyncAwait`, `asyncPhased` and `forasync`, each accepting a different number of their respective arguments.

The implementation of `asyncAt` is as shown in Figure 7. If the source and destination places are same, the lambda is scheduled as a local `async`. The other case is treated differently. A `std::function` class template (Figure 6) is a general purpose polymorphic function wrapper whose instances can store, copy and invoke lambdas. In the case of `asyncAt` both the lambda closure object and the `std::function` pointer are required for remote execution. The C++ compiler creates a separate `class` for each lambda function and so each lambda closure objects differs from each other. Thus, the `asyncAt` is templated to make the type of lambda available at runtime. There are three steps leading to the execution of a lambda remotely. *First*, the current `place` allocates memory in its partition of the global address space and copies the lambda object to it. *Second*, a UPC++'s `async` (asynchronous active message) taking the lambda as a parameter is invoked on the remote `place`. To ensure this UPC++ call is executed only by the communication worker (Section 4.2), the UPC++'s `async` is wrapped inside a lamda function and passed to the runtime. *Third*, the remote `place` processes the active message by executing the `asyncAt_wrapper` function which copies the lambda into the current `place` partition in the global address space and executes the lambda. On completion the reserved memory at both sender and destination `place` is deallocated. There is also some bookkeeping code to maintain the outgoing remote task count and incoming remote tasks count. They are explained in Section 4.3.

The implementation of `asyncCopy` is as shown in Figure 8. Similarly to `asyncAt`, `asyncCopy` is also scheduled to execute only at the communication worker and hence the UPC++ calls are wrapped inside a lambda function. Here the UPC++'s `async_copy` (asynchronous copy function) for remote copy is used. A callback function is also registered on this asynchronous copy, which gets invoked on the source `place` once the entire source data is copied to the destination buffer. Apart from the bookkeeping code, the callback is used to do a DDF_PUT if a DDF was supplied to the `asyncCopy`. It ensures that `asyncAwait` depending on those DDFs are notified when `asyncCopy` completes.

4.2 Integrating OCR with UPC++

We are using OCR (Section 2.2) as the Habanero-C++ work-stealing runtime and UPC++ (Section 2.4) for global address space abstraction for memory management and data communications. HabaneroUPC++ asynchronous task relies on HClib (Section 2.3) to translate to an EDT (Event-Driven Task) representation, which can be passed to OCR for scheduling. In this section we discuss the modifications made to integrate OCR with UPC++. Figure 9 depicts different components of HabaneroUPC++ and shows how HabaneroUPC++ application's executable is created.

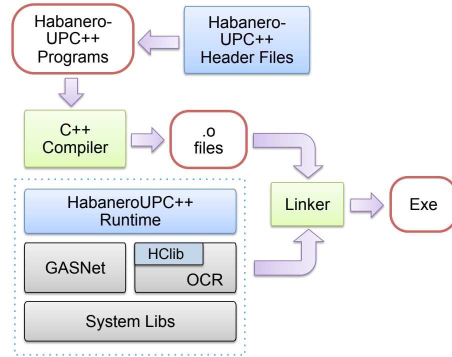


Figure 9: HabaneroUPC++ components.

We take the approach of Chatterjee et al. [7] and create a dedicated communication worker per HabaneroUPC++ place. Their experiments demonstrate that the benefits of a dedicated communication worker can outweigh the loss of parallelism from the inability to use it for computation. If OCR is configured to run on "`n`" cores then there will be one communication worker and "`n-1`" computation workers.

4.2.1 *Communication worker*

Other than the usual work-stealing double-ended queue (deque), the communication worker also maintains a semi-concurrent deque. These are respectively named in_deque and out_deque. The in_deque allows for concurrent push and steal operations. The out_deque only allows concurrent push operations and non-concurrent pop operations. Having two separate deques allows easy identification of local and remote tasks when communication worker pops tasks for remote transfers and pushes tasks for local executions.

The communication worker is responsible for invoking the `main` function of the HabaneroUPC++ application. Whenever local asynchronous tasks (`async`, `asyncAwait` and `asyncPhased`) are created, they are pushed to its in_deque. Remote asynchronous tasks (`asyncCopy` and `asyncAt`) are pushed to its out_deque. When the end of a `finish_spmd` block or the `main` function is reached, the communication worker pops tasks from its out_deque and executes them. If a task is received from a remote `place` it is pushed in its in_deque. The details of how the communication worker uses UPC++ to send and receive a remote task are discussed in Section 4.3.

4.2.2 *Computation worker*

When computation workers start, they have no tasks to execute and try to steal from other computation workers as well as the in_deque of the communication worker. Since the communication worker executes the `main` function, initially tasks are only available in its in_deque. Computation workers steal them from the in_deque and execute. These tasks can either be local or remote task. Whenever they encounter any local asynchronous tasks, they push it to their own deque. In case of remote asynchronous tasks, they always push to the out_deque of the communication worker and let the communication worker execute it. Hence, a push to out_deque is always potentially concurrent.

As an alternative to our current design, we can also allow a computation worker to start the `main` function rather than the commu-

nication worker. It can relieve the pressure on the in_deque of communication worker due to steals from multiple computation workers. However, due to help-first work-stealing scheduling policy of OCR, again all the tasks will be first queued at the deque of the computation worker starting the application. Allowing the communication worker to start the application also helps us in simplifying the single worker case, where the communication worker plays a dual role. Once it is unable to pop tasks from its out_deque, it will behave like a computation worker, stealing tasks from its in_deque and executing them.

4.3 Implementation of finish_spmd

The `finish_spmd` construct waits for all the dynamically spawned asynchronous tasks in its scope, which includes both local and remote tasks. The pseudo code for the runtime implementation of `finish_spmd` is shown on Figure 10. HabaneroUPC++ follows SPMD approach and hence the communication worker at each `place` is responsible to start the finish scope upon entering `finish_spmd`. It executes the lambda function and pushes all asynchronous tasks into the appropriate deques (Section 4.2.1). It then occasionally checks the quiescence of the finish scope, meaning all tasks have either completed locally or remotely at all `place`. The quiescence detection loop is composed of three parts; outgoing remote task processing, incoming remote task processing and a task count operation. The algorithm relies on two counters, one for outgoing remote tasks and another for incoming remote tasks. On a loop iteration, first the worker pops tasks from its out_deque and executes them. These tasks essentially contain UPC++ calls (Figures 7 and 8). The outgoing remote task counter was incremented before this task was pushed to the out_deque of communication worker (Figures 7 and 8). Second, the worker flushes the UPC++'s task queue. This removes all pending outgoing remote tasks from UPC++'s internal task queue and also receives incoming remote tasks (one at a time). When an incoming task is received, it is wrapped inside a local `async` and pushed to the in_deque for computation workers to steal and execute (Section 4.2.2). If this task contains another remote task (`asyncAt` or `asyncCopy`), the computation worker will push it to the out_deque of the communication worker. When an incoming remote task is executed, the incoming task counter is incremented by the computation worker (Figures 7 and 8). Third and last, the communication worker computes a local count by subtracting the sum of total of outgoing remote tasks and pending local tasks from the total incoming remote tasks. A global count of pending tasks for all `place` is obtained by performing a sum reduction using a UPC++ `allreduce` collective operation. If the global count equals zero, the communication worker exits the loop. Our approach to `finish_spmd` implementation is quite similar to the approach of Yang et al. [36]. The two key differences are: a) we are using blocking `allreduce` instead of their non-blocking `allreduce`; and b) before going into the `while` loop, the worker in their implementation would execute all the nested asynchronous tasks. However, the communication worker in HabaneroUPC++ follows help-first policy and returns just after pushing the outermost task into the deque.

5. PERFORMANCE EVALUATION

5.1 Experimental Setup

Before presenting the performance evaluation of HabaneroUPC++, we first describe our experimental setup.

5.1.1 Benchmarks

We have used two benchmarks from UPC++ distribution and ported them to HabaneroUPC++. These benchmarks are briefly described below. Their detailed description is available in [39].

SampleSort It sorts a large distributed array of 64-bit integer keys. We are using weak scaling and the total keys per UPC++ `place` are $48 \times 1024 \times 1024$. The keys are generated using `rand()` function.

LULESH This is a shock hydrodynamics proxy application. This benchmark restricts the number of UPC++ `place` to a perfect cube of an integer. This benchmark is also using weak scaling. In our experiments the inputs to this benchmark are: the length of cube mesh along sides as 38 and total iteration count as 100.

5.1.2 Hardware Platform

We used Edison supercomputer at NERSC for our experimental evaluations. This is a Cray XC-30 system with Intel Ivy Bridge CPUs and an Aries interconnect with DragonFly topology. Each node has two sockets and each socket has 12 cores.

5.1.3 Software Platform

HabaneroUPC++ We have our runtime implementation and the benchmarks publicly available at:
`http://habanero-rice.github.io/habanero-upc/`

OCR Git version xstack-intel_2013-09-06-2-gd687b14.

HClib Git version v0.3-6-g62496d1.

UPC++ Git version ver_0.1-51-g5f438c2.

GCC Version 4.9.0.

5.1.4 Measurements

Once the nodes are allocated on the Edison, we run each experiment ten times. We report the execution time as the mean of these ten invocations along with a 95% confidence interval based on a Student t-test.

5.2 Results

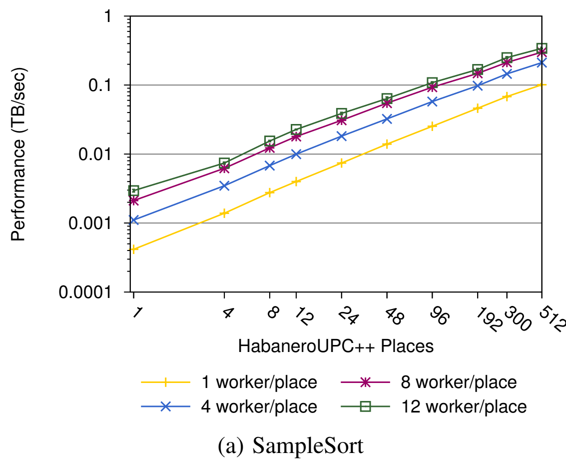
5.2.1 Work-Stealing Performance

Recall, HabaneroUPC++ uses Habanero-C++ work-stealing to achieve load-balance within a `place` and UPC++ to communicate across places. To measure the performance (weak scaling) of this programming model, we port our two benchmarks to HabaneroUPC++ and run them by varying total number of places and work-stealing workers. In this experiment (Figure 11) we launch one HabaneroUPC++ `place` per socket and vary the total number of work-stealing workers as 1, 4, 8 and 12 at each place. Y-axis represents the performance and X-axis shows total number of places. Both the axes are in log-scale.

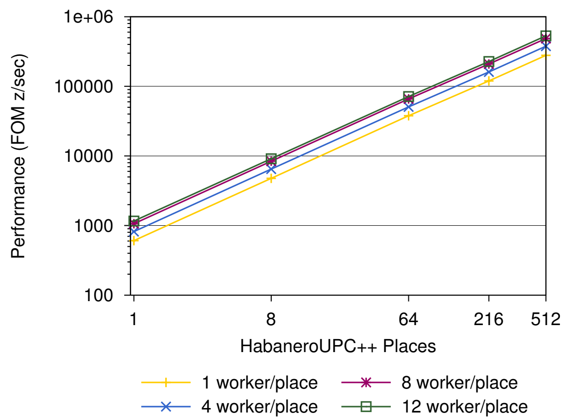
The UPC++ version of SampleSort uses `qsort` function from C++'s `cstdlib.h` header file for local sort of keys. There are total of three such occurrences of `qsort` functions. For porting to HabaneroUPC++ we take the default version of SampleSort and replace all the three `qsort` with a parallel divide and conquer implementation. This modified `qsort` uses `async` to parallelize each of the sub-problems and `finish` to join all these `async`. A threshold of 5120 is used to control the task granularity. This benchmarks also uses one `asyncCopy` to distribute and copy the array across the places. There is no `asyncAt` in the benchmark. `finish_spmd` is

```
void finish_spmd(std::function<void()>
                                  lambda) {
  // Start finish scope
  allocate_finish_object();
  // Execute the lambda containing
  // asynchronous tasks.
  lambda();
  // Loop until no more pending tasks
  // at global scope (both local and
  // remote)
  while(true) {
    // Pop and execute tasks from out_deque
    while(true) {
      void* task = pop_out_deque();
      if(task == NULL) break;
      else {
        // Execute lambda function in the task.
        // This lambda contains UPC++ calls,
        // details in Figure 7 and 8.
        async_wrapper(lambda); // see Figure 6
      } // end else
    } // end while
    // Send and receive remote tasks in
    // UPC++ queue
    void* incoming_remoteTask = advance_upcxx();
    if(incoming_remoteTask != NULL) {
      // Wrap it as local async which will
      // push this task to in_deque
      async([=]() {
        // Call UPC++ library to execute this task.
        execute_upcxx(incoming_remoteTask);
      }); // end async
    } // end if
    tasks_count = incoming_tasks - (outgoing_tasks
                 + total_local_pending_tasks);
    // Find total global pending tasks
    allreduce(&tasks_count,
                 &global_tasks_count, SUM);
    if(global_tasks_count==0) break;
  } // end while
  // end finish scope
  free_finish_object();
} // end finish_spmd
```

Figure 10: Runtime implementation of `finish_spmd`

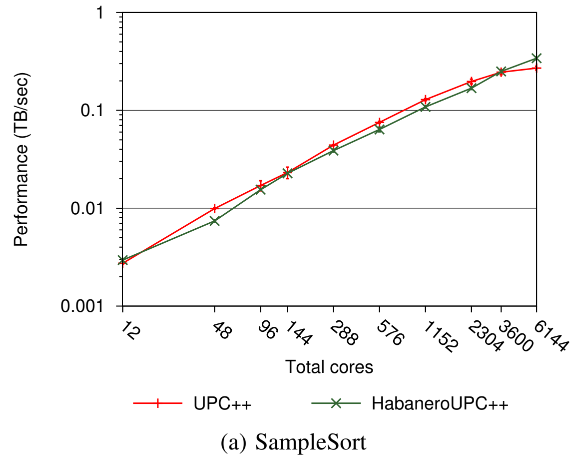


(a) SampleSort

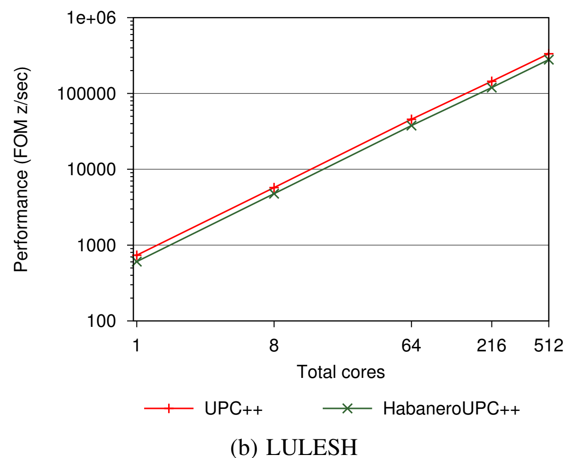


(b) LULESH (number of total `place` required to be a perfect cube by application)

Figure 11: Weak scaling performance using HabaneroUPC++ and varying number of work-stealing worker threads per `place`.



(a) SampleSort



(b) LULESH

Figure 12: Performance comparison of HabaneroUPC++ with UPC++.

used to join the asyncCopy task. The result of this experiment is shown in Figure 11(a). The performance of the benchmarks is represented on y-axis as total terabytes sorted per second (TB/sec). As we can see, increasing the total number of work-stealing worker threads increases the performance. We noticed an improvement of 3.4× by increasing the worker threads count from 1 to 12 with 512 places.

The UPC++ version of LULESH has several for-loops. We modify the default LULESH to use forasync instead of for-loops. The forasync tasks are joined with the help of finish. Default LULESH also uses UPC++ version of asynchronous copy and its own wait function. We modify LULESH to use our asyncCopy and finish_spmd to join them. There are total 3 such modifications. This benchmark too does not use asyncAt. The result of this experiment is shown in Figure 11(b). The performance of LULESH (FOM – zones per second) is represented on y-axis. We noticed consistent improvement of 2× by increasing the number of work-stealing worker threads count from 1 to 12 at all places.

5.2.2 HabaneroUPC++ Performance versus UPC++

The performance comparison of the HabaneroUPC++ version of benchmarks with the UPC++ version is shown in Figure 12. Y-axis shows the performance and X-axis shows the total number of cores used across each implementation. Both the axes are in log-scale.

For SampleSort benchmark, in the HabaneroUPC++ version we compare the performance of P places, each running with 12 work-stealing workers (total 12×P execution units) against the UPC++ version running with 12×P places. To ensure same computation size (weak scaling) across both versions, HabaneroUPC++ uses total keys per place as 12×N whereas UPC++ uses 1×N (N = 4×1024×1024). The result of this experiment is shown in Figure 12(a). Similar experimental setup is not viable for LULESH. In the HabaneroUPC++ version of LULESH we compare the performance of P places, each running with 1 work-stealing worker (total 1×P execution units) against the UPC++ version running with P places. The result of this experiment is shown in Figure 12(b).

The HabaneroUPC++ version of SampleSort performs nearly identical to UPC++. However, UPC++ version of LULESH consistently performs 20% better than HabaneroUPC++ version. This slight gap in performance is due to the overheads of work-stealing and heap allocation of C++11 lambda objects. Prior studies have shown that work-stealing overheads can be minimized by tweaking the compilers [34, 17]. As future work, we would like to develop similar techniques for Habanero-C++ work-stealing runtime. We would also like to include irregular benchmarks for performance study. Irregular applications pose significant challenges to achieving scalable performance on large-scale multicore clusters. These applications require dynamic load balancing to maintain efficiency. Prior studies have demonstrated that work-stealing implementations can provide very effective load-balancing for these kinds of applications [9, 8]. We predict that HabaneroUPC++ can definitely provide better performance for these irregular applications.

6. RELATED WORK

C++11 brings rich support for threading. It provides a function template std::async. This async takes callable object (or function) as an argument and returns a std::future object. This async can execute asynchronously. The user can use a get() function over the std::future object to wait for the async to complete and fetch the result of function execution. Habanero-C++ async differs greatly from C++11 async. Other than providing 3 different varieties of async, Habanero-C++ also allows arbitrary nesting of async. The user can join all the async using a single finish. Another great feature provided by C++11 is lambda functions. We are unaware of any C++ based dynamic tasking library which uses lambda functions as we do. Habanero-Java library [15] is a very recent, pure Java 8 library implementation of Habanero constructs. User interfaces to Habanero constructs are very similar across both Habanero-Java and Habanero-C++. However, the runtime implementations are very different. Being a C++ implementation, Habanero-C++ has the advantage that it can be combined with any C++ based high performance libraries.

Work-stealing is a very popular technique for load-balancing of dynamically spawned tasks and is used extensively. Chatterjee et al. designed HCMPI runtime, which is an integration of Habanero-C (Section 2.1) with MPI [7]. HCMPI unifies asynchronous task parallelism at intra-node level with MPI's message passing model at the inter-node level. However, by using MPI it's not able to harness the benefits of PGAS programming model. It's able to tie only the remote message transfer with the Habanero-C's finish-async constructs. By using a PGAS approach, HabaneroUPC++ offers better productivity and also provides asynchronous remote function shipping along with asynchronous remote copy. HCMPI requires complex compiler transformations unlike HabaneroUPC++. The approach of using a dedicated communication worker is similar in both HabaneroUPC++ and HCMPI, but HCMPI communication worker does not uses two deques as in HabaneroUPC++.

X10 and Chapel are very recent PGAS implementations. X10 introduced finish-async style programming model and uses work-stealing scheduling for load-balancing [32, 31, 38]. Both X10 and Chapel rely on compiler transformations to map user code to native code. Being a new programming language they are currently not as popular as C++. HabaneroUPC++ takes the C++ approach and adds finish-async style asynchronous tasking to SPMD PGAS programming model. By using C++11 lambda functions, HabaneroUPC++ avoids complex compiler transformations while retaining the elegance of language constructs. Unlike X10, HabaneroUPC++ currently does not support distributed work-stealing. The finish_spmd function in HabaneroUPC++ library is very similar to FINISH_SPMD pragma in X10 [31], although its implementation differs.

Min et al. introduced API based task library for using dynamic tasking in UPC [21]. The idea of favoring work-stealing inside PGAS programming model is similar across both Min et al. and HabaneroUPC++. However, there are several differences in the implementations. Some of them are: a) API based task library lacks the productivity of finish-async programming model; b) finish-async allows arbitrary nesting of both finish and async and provides more control to the programmer; and c) HabaneroUPC++ not just allow dynamic tasking but also allows creating dependencies among asynchronous tasks (asyncAwait and asyncPhased).

7. CONCLUSION

In this paper we presented HabaneroUPC++, a C++11 lambda functions-based compiler-free PGAS library. This library integrates Habanero's intra-place dynamic tasking constructs with UPC++'s inter-place asynchronous remote copy and asynchronous function shipping features. We also present a finish implementation for joining both local and remote asynchronous tasks. By using C++11 lambda functions we retain the syntactic convenience of language-based approaches while avoiding their associated complexities. Our intra-place work-stealing runtime uses a combination of communication and computation worker threads to enable the integration of the two programming models. We have presented a design based on a single communication worker run-

ning at each `place` that is responsible for managing the traffic of all the inter-`place` asynchronous tasks. This design allows the computation workers to execute both the local tasks as well as the tasks received from remote places. We have evaluated HabaneroUPC++ on Edison supercomputer at NERSC by using two benchmarks. We have scaled the benchmarks up to 6K cores and vary the total number of work-stealing worker threads at each `place` to demonstrate the performance and productivity of HabaneroUPC++.

There are several exciting future directions for this work. Some of them are a) distributed work-stealing implementation; b) extending the performance evaluation to a wider variety of benchmarks; c) a study of the limitations and overhead of using C++11 lambda functions and techniques to overcome them; and d) implementation of a non-SPMD (X10 style) HabaneroUPC++ and comparison with the SPMD approach.

8. REFERENCES

[1] K. Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems. *DARPA IPTO, Tech. Rep*, 15, 2008.

[2] B. Carlson, T. El-Ghazawi, B. Numrich, and K. Yelick. Programming in the partitioned global address space model. *Tutorial at Supercomputing*, 2003.

[3] V. Cavè. HClib: a library implementation of the Habanero-C language. `http://habanero-rice.github.io/hclib/`, 2013.

[4] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *PPPJ*, pages 51–61, 2011.

[5] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007.

[6] P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[7] S. Chatterjee et al. Integrating asynchronous task parallelism with MPI. In *IPDPS*, pages 712–725, 2013.

[8] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, pages 536–545, 2008.

[9] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *SC*, pages 53:1–53:11, 2009.

[10] T. El-Ghazawi and L. Smith. UPC: unified parallel C. In *SC*, 2006.

[11] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzarán, G. Almási, J. Moreira, and D. Padua. The hierarchically tiled arrays programming approach. In *LCR*, pages 1–12, 2004.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[13] M. Garland, M. Kudlur, and Y. Zheng. Designing a unified programming model for heterogeneous machines. In *SC*, 2012.

[14] M. Grossman, A. S. Sbîrlea, Z. Budimlić, and V. Sarkar. CnC-CUDA: Declarative programming for GPUs. In *LCPC*, pages 230–245, 2011.

[15] S. Imam and V. Sarkar. Habanero-java library: A Java 8 framework for multicore programming. In *PPPJ*, pages 75–86, 2014.

[16] J. Järvi and J. Freeman. C++ lambda expressions and closures. *Science of Computer Programming*, 75(9):762–772, 2010.

[17] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu. Work-stealing without the baggage. In *OOPSLA*, pages 297–314, 2012.

[18] D. Lea. A Java Fork/Join framework. In *JAVA*, pages 36–43, 2000.

[19] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA*, pages 227–242, 2009.

[20] B. Meister, N. Vasilache, D. Wohlford, M. Baskaran, A. Leung, and R. Lethin. *R-Stream Compiler*. Springer US, 2011.

[21] S. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *PGAS*, 2011.

[22] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *LFP*, pages 185–197, 1990.

[23] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[24] Compute unified device architecture programming guide. NVIDIA, 2007.

[25] Open Community Runtime. `https://01.org/open-community-runtime/`, Intel Open Source Technology Center, 2014.

[26] I. Patel and J. Gilbert. An empirical study of the performance and productivity of two parallel programming models. In *IPDPS*, pages 1–7, April 2008.

[27] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., first edition, 2007.

[28] Habanero-C Overview. `https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C`, Rice University, 2013.

[29] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *ICS*, pages 277–288, 2008.

[30] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS*, pages 1–12, 2009.

[31] O. Tardieu et al. X10 and APGAS at petascale. In *PPoPP*, pages 53–66, 2014.

[32] O. Tardieu, H. Wang, and H. Lin. A work-stealing scheduler for X10's task parallelism with suspension. In *PPoPP*, pages 267–276, 2012.

[33] S. Tasirlar and V. Sarkar. Data-driven tasks and their implementation. In *ICPP*, pages 652–661, 2011.

[34] K. Taura, K. Tabata, and A. Yonezawa. StackThreads/MP: Integrating futures into calling standards. In *PPoPP*, pages 60–71, 1999.

[35] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *LCPC*, pages 172–187, 2010.

[36] C. Yang, K. Murthy, and J. Mellor-Crummey. Managing asynchronous operations in Coarray Fortran 2.0. In *IPDPS*, pages 1321–1332, May 2013.

[37] K. Yelick et al. Titanium: A high-performance Java dialect. In *ACM*, pages 10–11, 1998.

[38] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. Saraswat, and M. Takeuchi. GLB: Lifeline-based global load balancing library in X10. In *PPAA*, pages 31–40, 2014.

[39] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: a PGAS extension for C++. In *IPDPS*, 2014.