Integrating Asynchronous Task Parallelism with MPI

Sanjay Chatterjee, Sağnak Taşırlar, Zoran Budimlić, Vincent Cavé,
Milind Chabbi, Max Grossman, Vivek Sarkar
Department of Computer Science
Rice University
Houston, USA
Email: {cs20, sagnak, zoran, vc8, mc29, jmg3, vsarkar}@rice.edu

Yonghong Yan
Department of Computer Science
University of Houston
Houston, USA
Email: yanyh@cs.uh.edu

***Abstract*—Effective combination of inter-node and intra-node parallelism is recognized to be a major challenge for future extreme-scale systems. Many researchers have demonstrated the potential benefits of combining both levels of parallelism, including increased communication-computation overlap, improved memory utilization, and effective use of accelerators. However, current "hybrid programming" approaches often require significant rewrites of application code and assume a high level of programmer expertise.**

Dynamic task parallelism has been widely regarded as a programming model that combines the best of performance and programmability for shared-memory programs. For distributed-memory programs, most users rely on efficient implementations of MPI. In this paper, we propose *HCMPI* (Habanero-C MPI), an integration of the Habanero-C dynamic task-parallel programming model with the widely used MPI message-passing interface. All MPI calls are treated as asynchronous tasks in this model, thereby enabling unified handling of messages and tasking constructs. For programmers unfamiliar with MPI, we introduce *distributed data-driven futures* (DDDFs), a new data-flow programming model that seamlessly integrates intra-node and inter-node data-flow parallelism without requiring any knowledge of MPI.

Our novel runtime design for HCMPI and DDDFs uses a combination of dedicated communication and computation specific worker threads. We evaluate our approach on a set of micro-benchmarks as well as larger applications and demonstrate better scalability compared to the most efficient MPI implementations, while offering a unified programming model to integrate asynchronous task parallelism with distributed-memory parallelism.

***Keywords*—MPI, asynchronous task parallelism, data flow, data-driven tasks, phasers**

I. Introduction

It is widely accepted that the road to exascale computing will require preparations for systems with $O(10^6)$ nodes and $O(10^3)$ cores per node [1], [2]. It is therefore critical to find software solutions that can effectively exploit this scale of combined inter-node and intra-node parallelism. One popular direction is to integrate asynchronous task parallelism with a Partitioned Global Address Space (PGAS) [3] model as exemplified by the DARPA HPCS programming languages (Chapel [4] and X10 [5]), and by recent multithreading extensions to established PGAS languages (UPC [6] and CAF [7]). PGAS programming models offer HPC programmers a single-level partition of a global address space with control of data-to-thread affinity/locality. While it has been shown that there are certain classes of applications for which the PGAS models are superior to MPI, MPI is still used extensively in large numbers of applications on the largest supercomputers in the world. Many obstacles still remain for the PGAS languages to surpass MPI in supporting these applications due to the overheads associated with maintaining a global address space, as well as the software engineering challenges of migrating MPI-based codes to PGAS.

On the other hand, harnessing $O(10^3)$-way parallelism at the intra-node level will be a major challenge for MPI programmers, for multiple reasons. The parallelism will have to exploit strong rather than weak scaling, since the memory per node is not increasing at the same rate as the number of cores per node. Programmers will also have to exploit heterogeneous processors and accelerators such as GPUs and FPGAs within a node. Finally, programs will have to be amenable to dynamic adaptive scheduling techniques in order to deal with non-uniform clock speeds and other load imbalances across cores that arise due to power management, fault tolerance, and other dynamic services.

We present a unified programming model for shared- and distributed-memory systems, with integrated support for asynchronous tasking and intra- and inter-node synchronization and communication. This programming model is positioned between two-level programming models (such as OpenMP+MPI), and single-level PGAS models. In this model, point-to-point communication tasks can be offloaded from the computation task's critical path, and unified primitives enable system-wide collective operations across tasks and MPI processes. We integrate intra-node asynchronous task parallelism with inter-node MPI communication to create a scalable non-blocking runtime system that supports both asynchronous task parallelism and data-flow parallelism. We have implemented this approach by extending the Habanero-C (HC) research language with MPI; the extended version is referred to as *HCMPI* throughout this paper[1].

For programmers unfamiliar with MPI, we introduce *distributed data-driven futures* (DDDFs), a new data-flow programming model that seamlessly integrates intra-node and inter-node data-flow parallelism without requiring any knowl-

[1]The HCMPI acronym was first introduced in a poster abstract [8]; however, the HCMPI system described in this paper is a complete redesign and re-implementation of the system outlined in [8].

edge of MPI. Since each DDDF has a globally unique id (guid) in a global name space, we refer to the DDDF model as an Asynchronous Partitioned Global Name Space (APGNS) programming model. In this model, distributed tasks form the basic building blocks for parallel computations. The tasks communicate via single-assignment distributed data items stored in the DDDFs. The APGNS model can be implemented atop a wide range of communication runtimes that includes MPI and GASNet; this paper includes results based on the use of MPI as the communication runtime for DDDFs.

A key assumption in the HCMPI runtime design is that it will be feasible to dedicate one or more cores per node to serve as *communication workers* in future many-core architectures. Thus, a program's workload can be divided into computation and communication tasks that run on computation and communication workers respectively. Our experimental results in Section IV show that even for today's multicore architectures, the benefits of a dedicated communication worker can outweigh the loss of parallelism from the inability to use it for computation. Further, the foundational synchronization constructs in our programming model such as *finish*, *phasers* and *data driven tasks* can be applied uniformly to computation tasks and communication tasks.

The rest of the paper is organized as follows. Section II summarizes the HCMPI programming model. We explain the details of our runtime system in Section III and evaluate HCMPI performance on two cluster parallel machines in Section IV. For the UTS benchmark on the ORNL Jaguar machine with 1024 nodes and 16 cores/node, HCMPI performed 22.3× faster than MPI for input size T1XXL and 18.5× faster than MPI for input size T3XXL (using the best chunking and polling parameters for both HCMPI and MPI). Section V summarizes related work, and Section VI contains our conclusions.

II. HCMPI Programming Model

This section describes the HCMPI programming model which unifies shared- and distributed-memory parallelism using Habanero-C task parallelism at intra-node level and MPI at inter-node level. We provide an overview of HCMPI constructs and refer to key APIs in brief. HCMPI APIs are discussed in greater detail in [9].

A. Node-level Task Parallelism

HCMPI uses the Habanero-C `async-finish` task programming model for exploiting intra-node parallelism. This model is based on the Habanero-Java [10] and X10 [5] task programming models, where tasks are created using the `async` construct, and synchronized using the `finish` construct. The statement `async` ⟨*stmt*⟩ causes the parent task to create a new child task to execute ⟨*stmt*⟩ *asynchronously* (i.e. before, after, or in parallel) with the remainder of the parent task. The statement `finish` ⟨*stmt*⟩, performs a join operation that causes the parent task to execute ⟨*stmt*⟩ and then wait until all the tasks created within ⟨*stmt*⟩ have terminated (including transitively spawned tasks). Figure 1 illustrates this concept by showing a code schema in which the parent task, T_0, uses an `async` construct to create a child task T_1. Thus, `STMT1` in task T_1 can potentially execute in parallel with `STMT2` in task T_0.

While Cilk `spawn` and `sync`, or the OpenMP `task` and `taskwait` constructs have similar syntax and effects, the `async/finish` constructs supports more general dynamic execution scenarios that are difficult to express in Cilk or OpenMP [11].

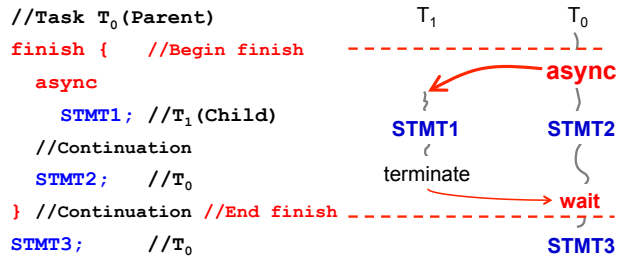


Fig. 1: An example code schema with `async` and `finish`

Any statement can be executed as a parallel task, including for-loop iterations and method calls. Figure 2 shows a vector addition example that uses `async` and `finish` constructs. We use loop chunking to allow each `async` task to perform the addition on a chunk of data. The `IN` keyword ensures that the task will have its own copy of the `i` variable, initialized to the value of `i` when the task is created. The semantics of `IN` is similar to that of the OpenMP `firstprivate` keyword.

HC supports *phasers* [12], [13] for fine-grained synchronization among dynamically created tasks. Phasers unify collective and point-to-point synchronization between tasks in a single interface, and are designed for ease of use and safety to help to improve programmer productivity in task parallel programming and debugging. The use of phasers guarantees two safety properties: deadlock-freedom and phase-ordering. These properties, along with the generality of its use for dynamic parallelism, distinguish phasers from other synchronization constructs such as barriers, counting semaphores and X10 clocks. In Habanero-C, tasks can register on a phaser in one of three modes: `SIGNAL_WAIT_MODE`, `SIGNAL_ONLY_MODE`, and `WAIT_ONLY_MODE`. The mode signifies a task's capabilities when performing synchronization operations on a specific phaser.

For data locality optimization, HC uses Hierarchical

```
int  PART_SIZE=16;
/* vector addtion: A + B = C, size is modular of 16 */
void vectorAdd(float * A, float * B, float * C, int size) {
int i, parts = size/PART_SIZE;
finish for (i=0; i < parts; i++) {
   async IN(i) {
      int j, start = i*PART_SIZE;
      int end = start + PART_SIZE;
         for (j=start; j < end; j++)
            C[j] = A[j] + B[j];
   }
}
```

Fig. 2: Task parallel programming using `async` and `finish`

Place Trees (HPTs) [14]. The runtime provides APIs for place identification, e.g `hc_get_current_place()` and `hc_get_parent_place()` return the current and parent places respectively. It allows the program to spawn tasks at places, which for example could mean cores, groups of cores with shared cache, nodes, groups of nodes, or other devices such as GPUs or FPGAs. The work-stealing scheduler executes tasks from the HPT with heuristics aimed to preserve locality. The HPT specification, an XML document, is optional for HC program execution. If an HPT is not specified, a single-level HPT is assumed by default. (This default was used for all results in this paper.)

HC supports creation of Data-Driven Tasks (DDTs) [15]. A DDT is a task that synchronizes with other tasks through full-empty containers named Data-Driven Futures (DDFs). A DDF obeys the dynamic single assignment rule, thereby guaranteeing that all its data accesses are race-free and deterministic. `DDF_CREATE()` is a function for creating a DDF object. The producer and consumer tasks use a pointer to DDF to perform `DDF_PUT()` and `DDF_GET()` operations. `DDF_PUT()` is the function for writing the value of a DDF. Since DDFs obey the single-assignment property, only one producer may set its value and any successive attempt at setting the value results in a program error. The `await` clause associates a DDT with a set of input DDFs: `async await` (ddf_a, ddf_b, ...) ⟨*stmt*⟩. The task cannot start executing until all the DDFs in its `await` clause have been put. `DDF_GET()` is a non-blocking interface for reading the value of a DDF. If the DDF has already been provided a value via a `DDF_PUT()` function, a `DDF_GET()` delivers that value. However, if the producer task has not yet performed its `DDF_PUT()` at the time of the `DDF_GET()` invocation, a program error occurs.

B. Point-to-Point Communication

HCMPI unifies the Habanero-C intra-node task parallelism with MPI inter-node parallelism. A HCMPI program follows the task parallel model within a node and MPI's SPMD model across nodes. Computation tasks have the ability to create asynchronous communication tasks, and achieve MPI's blocking semantics via Habanero-C's `finish` and `await` constructs. The runtime guarantees non-blocking execution of the computation workers. HCMPI will not introduce any deadlocks when extending from deadlock-free MPI code. The HCMPI APIs and types are very similar to MPI, making the initial effort of porting existing MPI applications to HCMPI extremely simple. Most MPI applications can be converted into valid HCMPI programs simply by replacing APIs and types that start with MPI_ by HCMPI_[2]. The only MPI feature that HCMPI does not currently support is the remote memory access (RMA), however that is straightforward to add to HCMPI and is a subject of future work.

Computation tasks initiate asynchronous *non-blocking* point-to-point communication via runtime calls to `HCMPI_Isend` and `HCMPI_Irecv`, as shown in Table I. They return a request handle object called `HCMPI_Request`, similar to `MPI_Request`. An important property of an `HCMPI_Request` object is that it can also be provided wherever an HC DDF is expected for data-driven execution. A non-blocking send or receive call returns a status object whose type is `HCMPI_Status`. This object is implicitly allocated within all APIs that return a status.

Point-to-Point API	Description
HCMPI_Send	blocking send
HCMPI_Isend	non-blocking send
HCMPI_Recv	blocking recv
HCMPI_Irecv	non-blocking recv
HCMPI_Test	Test for completion
HCMPI_Testall	Test all for completion
HCMPI_Testany	Test any for completion
HCMPI_Wait	Wait for completion
HCMPI_Waitall	Wait for all to complete
HCMPI_Waitany	Wait for any to complete
HCMPI_Cancel	Cancel outstanding communication
Collectives API	**Description**
HCMPI_Barrier	barrier synchronization
HCMPI_Bcast	broadcast
HCMPI_Scan	scan
HCMPI_Reduce	reduce
HCMPI_Scatter	scatter
HCMPI_Gather	gather
(HCMPI_All* variants supported)	
Phaser API	**Description**
HCMPI_PHASER_CREATE	hcmpi-phaser create
HCMPI_ACCUM_CREATE	hcmpi-accum create
next	phaser synchronization
accum_next	accumulator synchronization
accum_get	accumulated value
Runtime API	**Description**
HCMPI_REQUEST_CREATE	create request handle
HCMPI_GET_STATUS	query status
HCMPI_Get_count	get count of received data

TABLE I: HCMPI API

HCMPI allows a computation task to wait for the completion of a communication task. In the structured task parallel model, the `finish` scope provides a natural completion point for all tasks that were started within the scope. Figure 3 shows an example usage of the `finish` scope in a structured task parallel program. A computation task can create a point-to-point communication task, and asynchronously continue execution. It can create more communication tasks within the `finish` scope and wait at the end of the scope for them to complete. Thus, *blocking* point-to-point communication is achieved by placing the nonblocking primitive inside a `finish` scope. For example, the code on Fig. 3 implements the blocking receive in HCMPI.

In the HCMPI data-flow model, synchronization with communications can be achieved through the `await` clause. A computation task can declare a communication dependency by referring to a `HCMPI_Request` handle in its `await` clause. Figure 4 shows an example usage.

Another way to wait for the completion of a communication task is through `HCMPI_Wait` and its variants `HCMPI_Waitall` and `HCMPI_Waitany`. The computation task logically blocks at the `HCMPI_Wait` for the asyn-

[2]While this replacement can be easily automated by a preprocessor or by API wrappers, we use the HCMPI_ prefix in this paper to avoid confusion with standard MPI.

```
finish {
  HCMPI_Irecv(recv_buf, ···);
  ··· //do asynchronous work
} // Irecv must be complete after finish
```

Fig. 3: Using the `finish` construct in HCMPI. A `finish` around `HCMPI_Irecv`, a non-blocking call, implements `HCMPI_Recv`, a blocking call.

```
HCMPI_Request * r;
HCMPI_Irecv(recv_buf, ···, &r);
async AWAIT(r) IN(recv_buf) {
  //read recv_buf
}
··· //do asynchronous work
```

Fig. 4: HCMPI Await Model

chronous communication task to complete. The synchronization event is provided by a `HCMPI_Request` handle and returns a `HCMPI_Status` object. Figure 5 shows an example of using `HCMPI_Status` to get a count of the number of elements received in a buffer after the completion of a `HCMPI_Irecv` operation.

```
HCMPI_Request * r;
HCMPI_Irecv(recv_buf, ···, &r);
··· //do asynchronous work
HCMPI_Status * s;
HCMPI_Wait(r, &s);
int count;
HCMPI_Get_count(s, HCMPI_INT, &count);
if (count > 0) { //read recv_buf }
```

Fig. 5: HCMPI Wait and Status Model

C. Collective Operations

HCMPI supports collective operations at the intra-node and inter-node levels We first discuss HCMPI synchronization at only the inter-node level, and then discuss the combined inter- and intra-node model.

Inter-node-only collective operations in HCMPI are similar to MPI collectives. Table I includes a partial list of supported HCMPI collectives. All HCMPI collective operations follow the blocking semantics discussed in Section II-B. We will add support for non-blocking collectives to HCMPI once they become part of the MPI standard. Figure 6 shows how to perform an inter-node-only barrier. In this example, asynchronous task `A()` is created before the barrier and can logically run in parallel with the barrier operation. However, function call `B()` must be completed before the barrier, and function call `C()` can only start after the barrier.

```
async A();
B();
HCMPI_Barrier();
C();
```

Fig. 6: HCMPI Barrier Model

One of the novel contributions of HCMPI is that it provides unified semantics for system wide collective operations. We combine inter-node MPI collectives with intra-node `phaser` synchronization into a new construct called *hcmpi-phaser*. An instance of `hcmpi-phaser` is created using the `HCMPI_PHASER_CREATE` API shown in Table I. The API accepts a registration mode argument, same as the `phaser` registration modes mentioned in Section II-A. Tasks registered on a `hcmpi-phaser` instance can synchronize both within the node and across nodes using the synchronization primitive `next`. Dynamic registration and deregistration is allowed, as well as arbitrary mode patterns. The inter-node SPMD model requires that every rank process creates its own `hcmpi-phaser` before participating in the global `next` operation. Figure 7 shows an example of using the `hcmpi-phaser` as a barrier.

```
finish {
  phaser *ph;
  ph = HCMPI_PHASER_CREATE(SIGNAL_WAIT_MODE);
  for (i = 0; i < n; ++i) {
    async phased(ph) IN(i) {
      ···; next;
      ··· //do post-barrier work
    } /*async*/ } /*for*/ } /*finish*/
```

Fig. 7: HCMPI Phaser Barrier Model

The HCMPI model integrates intra-node `phaser` accumulators [16] with inter-node MPI reducers using the *hcmpi-accum* construct. An instance of `hcmpi-accum` is created using the `HCMPI_ACCUM_CREATE` API. In this model, computation tasks at the intra-node level register on a `hcmpi-accum` instance and participate in the specified global reduction operation via the runtime call `accum_next(value)`, which takes as an argument the individual datum provided by the task for the reduction. By default, all tasks are registered in the `SIGNAL_WAIT_MODE`. Tasks arrive at the synchronization point with a value and participate to all `hcmpi-accum` instances they are registered with. After synchronization completes, `accum_get` will return the globally reduced value. At the inter-node level, we currently only support the `MPI_Allreduce` model. This means that a call to `accum_get()` will return the globally reduced value. Figure 8 shows an example of the `hcmpi-accum` model for the SUM operation.

```
finish {
 phaser *ph;
 ph = HCMPI_ACCUM_CREATE(HCMPI_SUM,HCMPI_INT);
 for (i = 0; i < n; ++i) {
 async phased IN(···) {
  int* my_val = get_my_val();
  accum_next(my_val);
  ···;} /*async*/ } /*for*/ } /*finish*/
  int* result = (int*)accum_get(ph);
```

Fig. 8: HCMPI Phaser Accumulator Model

D. Distributed Data Flow Model

We introduce *distributed* data-driven futures (DDDF) for inter-node parallelism, as a new extension to the intra-node

DDFs introduced in Section II-A. DDDFs enable unconstrained task parallelism at the inter-node level, without concerning the user about details of inter-node communication and synchronization. Thus, DDDFs can even be used by programmers who are non-experts in standard MPI. A DDDF includes a *node affinity* for every DDF object. The API `DDF_HANDLE(guid)` creates a handle on a DDDF identified by `guid`, a user managed globally unique id for the DDDF. The user provides two callback functions for the HCMPI runtime called `DDF_HOME(guid)` and `DDF_SIZE(guid)`. These functions should respectively provide a mapping from a `guid` to a DDF's *home* rank and the *put* data size, and should be available on all nodes. DDDFs provide support for the `put`, `get` and `await` operations using the API from the intra-node HC model at the inter-node level.

Let us consider the Smith-Waterman local sequence alignment benchmark in Fig. 9 as a DDDF example; its parallelism structure is discussed later in Fig. 23. The code in Fig. 9 implements a distributed-memory data-driven version of the benchmark. The only change from a shared-memory version is the use of `DDF_HANDLE` instead of `DDF_CREATE`, and the creation of user-provided `DDF_HOME` and `DDF_SIZE` function definitions. The `DDF_HOME` macro in this example performs a cyclic distribution on the global id, which enforces a row-major linearization of the distributed 2D matrix.

```
#define DDF_HOME(guid) (guid%NPROC)
#define DDF_SIZE(guid) (sizeof(Elem))
DDF_t** allocMatrix(int H, int W) {
  DDF_t** matrix=hc_malloc(H*sizeof(DDF_t*));
  for (i=0;i<H;++i) {
    matrix[i]=hc_malloc(W*sizeof(DDF_t));
    for (j=0;j<W;++j) {
      matrix[i][j] = DDF_HANDLE(i*H+j);
    }/*for*/}/*for*/
  return matrix;
}
DDF_t** matrix2D=allocMatrix(height,width,0);
doInitialPuts(matrix2D);
finish {
 for (i=0,i<height;++i) {
  for (j=0,j<width;++j) {
   DDF_t* curr  = matrix2D[ i ][ j ];
   DDF_t* above = matrix2D[i-1][ j ];
   DDF_t* left  = matrix2D[ i ][j-1];
   DDF_t* uLeft = matrix2D[i-1][j-1];
   if ( isHome(i,j) ) {
    async AWAIT (above, left, uLeft) {
     Elem* currElem = init(DDF_GET(above),
              DDF_GET(left),DDF_GET(uLeft));
     compute(currElem);
     DDF_PUT(curr, currElem);
}/*async*/}/*if*/}}/*for,for*/}/*finish*/
```

Fig. 9: Simplified Smith-Waterman implementation

III. HCMPI Implementation

The HCMPI runtime is a novel design based on dedicated computation and communication workers in a work-stealing scheduler, shown in Fig. 10. The HCMPI runtime has to create at least one communication worker per MPI-rank. The number

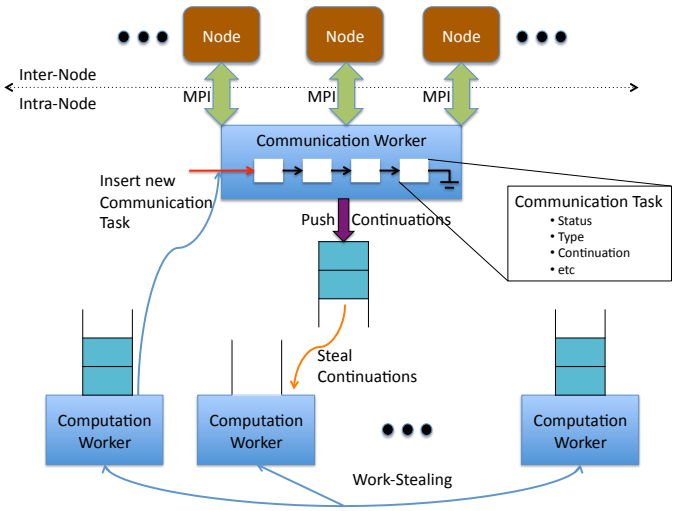


Fig. 10: The HCMPI Intra-node Runtime System

Fig. 11: Lifecycle of a Communication Task

of computation workers can be set at runtime by the `-nproc` command line option. Our experiments show that the benefits of a dedicated communication worker can outweigh the loss of parallelism from the inability to use it for computation. We believe that this trade-off will be even more important in future extreme scale systems, with large numbers of cores per node, and an even greater emphasis on the need for asynchrony between communication and computation.

The HCMPI runtime is an extension of the Habanero-C work-stealing runtime. Computation workers are implemented as pthreads (typically one per hardware core/thread). Each worker maintains a double-ended queue (deque) of lightweight computation tasks. A worker enqueues and dequeues tasks from the tail end of its deque. Idle workers steal tasks from the head end of the deques of other workers. A communication optimization scheme, such as the one implemented in [17], will be a natural extension to our implementation of HCMPI workers.

The HCMPI communication worker is dedicated to execute MPI calls, using a worklist of communication tasks implemented as a lock-free queue. Figure 11 shows the lifecycle of a communication task. When a computation worker makes an HCMPI call, it creates a communication task in the `ALLOCATED` state. The task is either recycled from the set of `AVAILABLE` tasks, or it is newly allocated and enqueued into the worklist. The task structure is initialized with required information, such as buffer, type, etc. and then set as `PRESCRIBED`. When the communication worker finds a `PRESCRIBED` task, it either issues an asynchronous MPI call for point-to-point communication or blocks for a collective MPI call. For asynchronous calls, the worker sets the task state as `ACTIVE` and moves on to the next task in the worklist. The worker tests `ACTIVE` tasks for completion using `MPI_Test`.

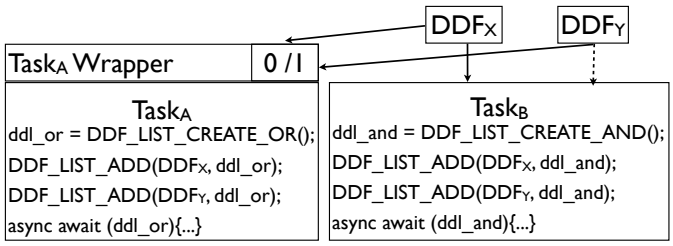


Fig. 12: HCMPI DDF Runtime

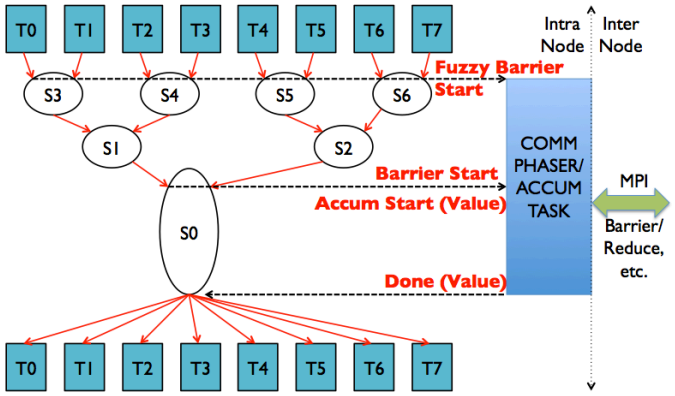


Fig. 13: HCMPI Phaser Barrier

Once an MPI operation has completed, the task state is set to `COMPLETED`. If the task is the last one to complete in the enclosing `finish` scope, the communication worker pushes the continuation of the `finish` onto its deque to be stolen by computation workers.

HCMPI implements event-driven task execution using Habanero-C's Data-Driven Tasks (DDTs) and Data-Driven Futures (DDFs), introduced in Section II-A. A `HCMPI_Request` handle is implemented as a DDF. Computational tasks created using `async await(req)`, where `req` is the `HCMPI_Request` handle, will start executing once the communication task represented by the handle has been completed. Further, `HCMPI_Wait` is implemented as `finish async await(req);`. `HCMPI_Waitall` and `HCMPI_Waitany` are implemented as extensions to `HCMPI_Wait` where a task waits on a list of DDFs. The key difference is that the *waitall* list is an AND expression while the *waitany* list is an OR expression. The handling of an AND list is similar to the one described in [15]. In case of the OR list, the runtime iterates over the list of DDFs found in the `await` clause. If a DDF is found to have been satisfied by a `put`, the task becomes ready for execution immediately. If no satisfied DDF is found, the task gets registered onto all DDF's on the list. When a `put` finally arrives on any of the DDF's the task get released and is pushed into the current worker's deque. To prevent concurrent `puts` from releasing the same task with an OR DDF list, each task contains a wrapper with a token bit to indicate if the task has already been released for execution, as shown in Fig. 12. This token is checked and set atomically to ensure the task is released only once. The HCMPI communication runtime is itself a client of the DDF runtime. It uses DDFs to communicate `MPI_Status` information to the computation tasks via a `DDF_PUT` of the `HCMPI_Status` object on to the `HCMPI_Request` DDF. `HCMPI_GET_STATUS` internally implements a `DDF_GET`.

A. Phaser Synchronization Model Implementation

HCMPI builds on Habanero-C's tree-based implementation of `phasers` to integrate inter- and intra-node synchronization. Tree based phasers have been shown to scale much better than flat phasers [13], [18]. `HCMPI_PHASER_CREATE` creates a phaser barrier, while `HCMPI_ACCUM_CREATE` creates an accumulator object. Tasks can dynamically register to and drop from a `hcmpi-phaser`. The `next` statement and the `accum_next` APIs act as the global synchronization points for barriers and accumulators. Figure 13 illustrates the synchronization process for HCMPI phaser barrier and accumulator operations. In the case of a barrier, tasks, T0 to T7, arrive at the `next` statement and signal the phaser. Then they start the wait phase. These tasks traverse the internal nodes of the phaser tree to see if they can become sub-masters at any of the sub-phaser nodes S0 to S6. The first task to arrive at a sub-phaser becomes the sub-master for that node. The sub-master collects the signals from its sub-tree and then signals its parent. This way, signals on the phaser tree propagate up to the root node. The first task to arrive at the root node becomes the phaser master. Others wait for the phaser master to signal the start of the next phase. HCMPI supports both relaxed (fuzzy) and strict barrier modes. In the fuzzy barrier mode, the first task to arrive at the wait phase on the leaf sub-phaser signals the phaser communication task to start the `MPI_Barrier` operation. This ensures overlapped inter-node and intra-node barrier operation. In the strict barrier mode, the `MPI_Barrier` operation is started only after the phaser master at the root sub-phaser receives all signals in the phaser tree. The phaser master waits on a notification from the communication task that the `MPI_Barrier` operation is completed. Once the notification arrives, the phaser master signals all the intra-node tasks to start their next phase. In case of phaser accumulators, each task arrives at the `accum_next` synchronization point with a value in addition to the signal. The value gets reduced to a single element at the root of the phaser tree and then the phaser master signals the hcmpi phaser communication task to start the `MPI_Allreduce` operation. The globablly reduced value is saved in the phaser data structure and can be retrieved by the `accum_get` call on that phaser object.

B. Distributed Data Flow Model Implementation

Distributed data-driven futures, introduced in Section II-D, are created through the `DDF_HANDLE` interface. The user-provided `DDF_HOME` function is used by the creation routine to identify it as a local or remote object. The call always returns a local handle. The DDF *home* has the responsibility of transferring data to remote requesters `await`-ing on that DDF. After the `put` completes, the *home* first completes the data transfer to all requests that have already arrived. Then

the communication worker starts a *listener* task to respond to future requests. The HCMPI runtime executes a global termination algorithm to take down all the *listener* tasks at the end of the program.

A data-driven task with an `async await` on a remote DDF registers on the local copy of the DDF handle. After the first DDT registers, the runtime sends the *home* location a message to register its intent on receiving the `put` data. The runtime also allocates a local buffer, and waits for the data response from *home*. The remote node always keeps a locally cached copy after the data arrives so that every subsequent `await` can immediately succeed. The dynamic single assignment property of DDFs ensures the validity of this local copy, and that the data transfer from *home* to remote happens at most once.

IV. Experimental Results

In this section we present experimental results of HCMPI performance on micro-benchmarks and strong scaling applications. For micro-benchmark performance we use standard test suites for multi-threaded performance and collective synchronization. We conduct our strong scaling experiments on UTS, a tree-based search application, and SmithWaterman, a heavily used sequence alignment algorithm in biomedical applications. We compare our performance against existing reference codes.

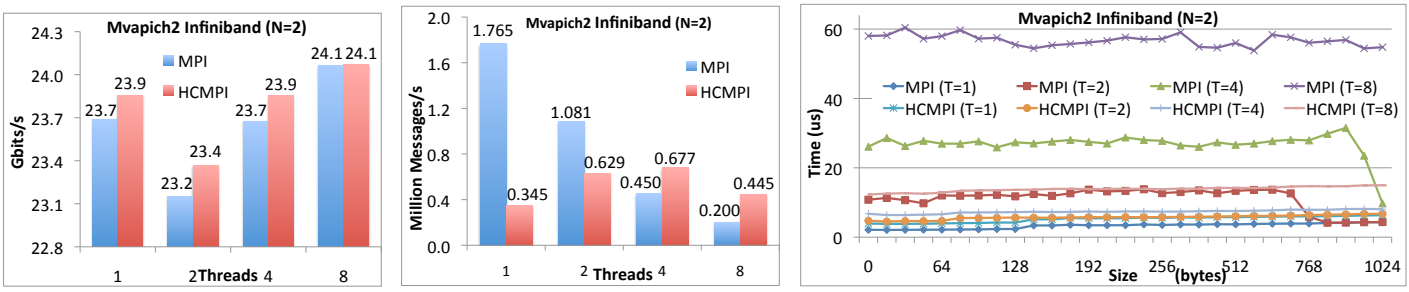
Our experimental framework uses the Jaguar supercomputer at Oak Ridge National Labs and the DAVinCI cluster at Rice University. The Jaguar supercomputer is a Cray XK6 system with 18,688 nodes with Gemini interconnect. Each node is equipped with a single 16-core AMD Opteron 6200 series processor and 32 GB of memory. For our experiments, we scaled up to 1024 nodes (16,384 cores) and used the default MPICH2 installation. The DAVinCI system is an IBM iDataPlex consisting of 2304 processor cores in 192 Westmere nodes (12 processor cores per node) at 2.83 GHz with 48 GB of RAM per node. All of the nodes are connected via QDR InfiniBand (40 Gb/s). On DAVinCI, we used up to 96 nodes (1152 cores) and used MVAPICH2 1.8.1.

A. Micro-benchmark Experiments

Many current hybrid models use MPI with Pthreads or OpenMP to expose a combination of threads and processes to the user. Such a model would have to deal with concurrent MPI calls from multiple threads implying that MPI has to operate in one of its multi-threaded modes. For our first set of micro-benchmark experiments, we used a test suite developed at ANL for multi-threaded MPI [19]. The benchmark programs initialize MPI using `MPI_THREAD_MULTIPLE` and intra-process multi-threading is achieved using `pthreads`. A *bandwidth* test is performed by measuring delays caused by sending large (8Mbyte) messages with low frequency. A *message rate* test transmits empty messages with high frequency. In the *latency* test, 1000 sends and 1000 receives are performed for different message sizes ranging from 0 to 1024. The average time delay for each size is reported. The HCMPI version creates as many computation workers as there are `pthreads` in the MPI version. HCMPI can use `MPI_THREAD_SINGLE` since all MPI calls are made by a single communication worker per process. Parallel tasks on multiple computation workers can communicate concurrently through the communication worker. This avoids using multi-threaded MPI, which typically performs worse than single-threaded MPI due to added synchronization costs.

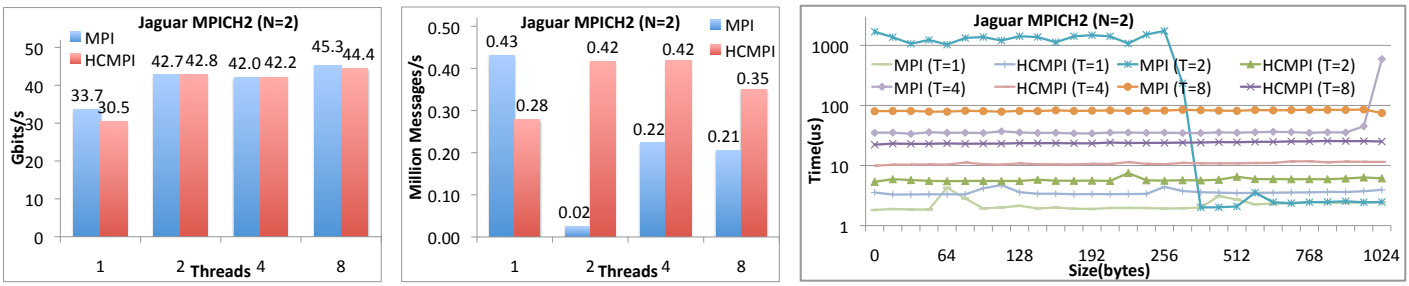
The micro-benchmark programs use two communicating processes. In our experiments, they are placed on two different nodes. The results in Fig. 14 are for MVAPICH2 with Infiniband on DAVinCI, and the results in Fig. 15 are for MPICH2 with Gemini on Jaguar. The bandwidth experiments in both cases show MPI and HCMPI performing close to each other. Since this test performs small number of communications with high data volume, the number of threads have a minimal impact. The message rate tests on the other hand sends a large number of low data volume messages. In this case, HCMPI starts performing better than multi-threaded MPI when we scale up the number of threads inside the process. We conclude it reflects higher synchronization overheads for communication inside multi-threaded MPI processes. The latency tests confirm our conclusions for message rate by showing that HCMPI latencies scale more gracefully than those for MPI when increasing the number of threads. The Jaguar message rate test shows a dip in performance when using 2 threads. This phenomenon was consistently repeatable over multiple runs of the benchmark. This fact is also reflected on the latency chart where we see that latency in MPI with 2 threads is an order magnitude higher than MPI with 8 threads.

For our next set of benchmarks, we measure the performance of HCMPI `phaser` barriers and accumulators. We compare against MPI-only and hybrid MPI+OpenMP performance. We used a modified version of the EPCC Syncbench [20] for barrier and reduction (accumulator) tests. The benchmarks run a loop containing a barrier or reduction operation for a large number of times. The cost of synchronization is estimated by subtracting the loop overhead from the iterations. We measure synchronization performance on 2 to 64 nodes while using 2 to 8 cores per node. In our experiment, the MPI-only version uses `MPI_THREAD_SINGLE`, while the hybrid MPI+OpenMP version uses `MPI_THREAD_MULTIPLE`. Both the HCMPI and hybrid versions use one process per node and use threads for the number of cores used inside a node. The MPI-only version uses one process for every core used in the experiment to perform distributed collective operations. The HCMPI test creates the number of tasks and computation workers equal to the number of cores used per node in the experiment. Together they perform the integrated synchronization at the intra-node and inter-node level for both barriers and accumulators. We measure HCMPI phaser barrier performance for both strict and fuzzy modes, introduced in Section III-A. The MPI+OpenMP hybrid version creates a parallel region of number of threads equal to the number of cores. In the strict barrier mode, threads first synchronize using a OpenMP barrier, then the `MPI_Barrier` is called by a `single` thread while the others wait at subsequent OpenMP barrier. The



(a): Bandwidth (b): Message Rate (c): Latency Micro-benchmark

Fig. 14: Thread Micro-benchmarks for MVAPICH2 on Rice DAVinCI cluster with Infiniband interconnect



(a): Bandwidth (b): Message Rate (c): Latency Micro-benchmark

Fig. 15: Thread Micro-benchmarks for MPICH2 on Jaguar Cray XK6 with Gemini interconnect

fuzzy barrier mode does not use the first OpenMP barrier. The hybrid reduction test completes a global reduction by first performing an OpenMP for loop reduction over the number of threads followed by `MPI_Allreduce` by a `single` thread. Remaining threads wait at a OpenMP barrier.

The results in Table II clearly demonstrate that MPI and hybrid times increase at a faster rate compared to HCMPI with increasing number of cores per node, for both barriers and accumulators. We also see that fuzzy barriers are much faster than strict barriers because of overlapped intra- and inter-process synchronization. HCMPI depends on MPI performance for inter-node synchronization. When scaling up the number of cores within a node, HCMPI is able to use intra-node phaser synchronization, while MPI depends on `MPI_Barrier` and `MPI_Allreduce` over all cores. Overall, hybrid MPI+OpenMP outperforms MPI while HCMPI outperforms both. These experiments were performed using MVAPICH2 on the DAVinCI cluster. We have not included the results for Jaguar because we discovered inconsistent `MPI_Barrier` performance with MPICH2.

B. Case Study: UTS

For our scaling experiment we chose the Unbalanced Tree Search (UTS) application [21], [22] from a publicly available source [23]. The UTS tree search algorithm is parallelized by computing the search frontier nodes in parallel. The search typically leads to unbalanced amount of work on parallel resources, which can then benefit from load balancing techniques. The reference MPI implementation of the benchmark, used as the baseline for creating the HCMPI version, performed parallel search using multiple MPI processes, and load balancing using inter-process work-sharing or work-stealing algorithms. In our experiments we have focused on the work-stealing version due to better scalability [22]. We scale our

Collective Synchronization Times in micro-seconds						
Nodes	2			4		
Cores	2	4	8	2	4	8
MPI Barrier	3.0	4.1	5.1	5.8	6.7	7.6
MPI+OMP Barrier (S)	2.5	2.8	3.9	5.0	5.8	6.7
HCMPI Phaser (S)	2.1	2.2	2.7	4.8	4.8	5.4
MPI+OMP Barrier (F)	2.6	2.9	3.7	4.9	5.2	6.1
HCMPI Phaser (F)	2.1	2.2	2.1	5.1	5.1	5.0
MPI Reduction	3.8	4.6	5.2	6.3	7.2	7.9
MPI+OMP Reduction	3.1	3.6	4.9	5.4	5.9	7.2
HCMPI Accumulator	2.6	2.8	3.5	4.9	5.0	5.8
Nodes	8			16		
Cores	2	4	8	2	4	8
MPI Barrier	9.1	9.8	11.1	12.6	13.4	14.7
MPI+OMP Barrier (S)	8.2	9.1	10.0	11.6	12.6	14.2
HCMPI Phaser (S)	7.7	7.7	8.6	11.3	11.2	12.1
MPI+OMP Barrier (F)	7.3	8.1	8.8	10.1	11.1	12.4
HCMPI Phaser (F)	7.5	7.5	7.6	10.9	10.7	10.8
MPI Reduction	9.5	10.7	12.1	12.8	14.3	15.3
MPI+OMP Reduction	8.2	9.1	10.5	11.1	12.4	14.1
HCMPI Accumulator	7.7	7.8	9.4	10.7	10.5	12.3
Nodes	32			64		
Cores	2	4	8	2	4	8
MPI Barrier	20.0	19.9	21.6	25.3	25.7	26.2
MPI+OMP Barrier (S)	17.2	19.0	20.8	21.8	24.7	26.2
HCMPI Phaser (S)	17.2	17.8	18.0	22.0	21.7	23.6
MPI+OMP Barrier (F)	13.5	14.5	16.6	19.4	20.8	24.0
HCMPI Phaser (F)	14.7	14.3	14.8	19.3	18.7	18.7
MPI Reduction	17.7	18.7	19.8	25.0	25.7	26.7
MPI+OMP Reduction	15.1	16.9	18.9	20.8	23.4	25.8
HCMPI Accumulator	14.7	15.4	16.9	20.8	20.6	23.5

TABLE II: EPCC Syncbench with MVAPICH2 on Infiniband. *Abbrv.* (S): Strict barrier (F): Fuzzy barrier

experiment up to 16,384 cores on the Jaguar supercomputer.

The HCMPI implementation of UTS adds intra-process parallelization to the reference MPI implementation. It does not modify the inter-process peer-to-peer work-stealing load balancing algorithm. HCMPI's goal is to benefit from shared-memory task parallelism on a node and uses only one process per node. In this context, inter-node and inter-process can be used interchangeably. In the HCMPI implementation a task has access to a small stack of unexplored nodes local to the worker thread it is executing on. When the stack fills up, nodes are offloaded to a deque for intra-node work-stealing. This strategy generates work for intra-node peers before it sends work to global peers. The use of non-synchronized thread-local stacks is for superior performance over deques. Global communication is handled by the communication worker. The HCMPI runtime uses a *listener* task for external steal requests while the computation workers are busy. When another process requests a steal, the listener task looks for internal work, trying to steal from the local work-stealing deques. If the local steal was successful, it responds with that work item, if not, with an *empty* message. Inside a node, when a computation worker runs out of work and is unable to steal work from local workers, it requests the communication worker to start a global steal. A global steal uses the reference MPI inter-process steal algorithm. During a global steal, if an active local computation worker has been able to create internal work, then some idle computation workers may get back to work. Once the communication worker receives a globally stolen work item, it pushes that item onto its own deque to be stolen by idle computation workers. Finally, the communication worker participates in a token passing based termination algorithm, also used in the reference MPI code.

In our experiments, we use two UTS tree configurations, T1XXL and T3XXL. T1XXL uses a geometric distribution and has a tree size of about 4 billion nodes. T3XXL uses a binomial distribution and has a tree size of 3 billion nodes. We varied the number of nodes from 4 to 1024 and cores per node from 1 to 16 in our experiments. To identify the best performing UTS configurations on Jaguar, we explored various chunk sizes, $-c$, and polling intervals, $-i$, on 64 nodes with 16 cores on both MPI and HCMPI for T1XXL as well as T3XXL. The best configuration of MPI for T1XXL was $-c = 4, -i = 16$, while for T3XXL was $-c = 15, -i = 8$. These configurations performed better on Jaguar than the published configurations in [22]. The best HCMPI configuration for T1XXL was $-c = 8, -i = 4$, while for T3XXL was again $-c = 8, -i = 4$. Finding the best UTS chunk size and polling intervals for each node and cores per node combination is outside the scope of this work. In our experiments, we allocate the same number of resources for both MPI and HCMPI. This means HCMPI runs one fewer computation worker than MPI because it dedicates one thread as communication worker. E.g. When using 4 nodes with 16 cores per node, MPI runs $4 \times 16 = 64$ processes, whereas HCMPI runs $4 \times 15 = 60$ computation workers and 4 communication workers, one per node. The MPI implementation uses `MPI_THREAD_SINGLE`.

Figures 16 and 18 respectively show the running time of MPI and HCMPI for the T1XXL workload. Similarly, Fig. 17 and 19 show the running time of MPI and HCMPI for the T3XXL workload. Individual lines show the performance for different number of cores per node. For MPI, each extra core amounts to an extra MPI process per node, whereas for HCMPI it amounts to an extra thread in the process on that node. For T1XXL, MPI stops scaling after approximately 4096 cores, and starts degrading rapidly. On the other hand, HCMPI keeps scaling perfectly to about 8192 cores and hardly degrades from then onwards. Results for T3XXL also show similar trends.

Figures 20 and 21 compare performance of HCMPI with MPI on T1XXL and T3XXL respectively. The peak performance improvement is about $22.3\times$ for 1024 nodes with 16 cores per node. A distinct crossover point in performance can be noticed in favor of HCMPI when the number of cores per node scales up. At 2 or 4 cores per node, HCMPI suffers from lack of parallel workers compared to MPI. But, as we scale up to 8 and 16 cores per node, HCMPI outperforms MPI.

To analyze this result further, we profiled both MPI and HCMPI codes using the built-in performance counters in the UTS application. First, the overall execution time is split into the following components: *work*, *overhead*, *search* and *idle*. *Work* represents the actual time spent on computation, that is, exploring nodes in the search tree. *Overhead* represents the time spent on making progress for others with global communication. MPI computation workers interrupt *work* every polling interval for this. *Search* represents the time spent trying to globally locate work. MPI workers enter this mode once they completely run out of work. HCMPI workers on the other hand, overlap this with computation as soon as the first worker cannot find local work. *Idle* time is the time spent in startup and termination. This is irrelevant for our comparison as we use the same startup and termination algorithms in both MPI and HCMPI. All times are reported as the average over all computing resources in the program. Next, we also profiled the total number of failed steal requests during program execution. This number represents the total amount of redundant communication in the system.

Table III provides statistical data for 3 node configurations: 64, 256 and 1024. We chose these 3 nodes as being representative of three regions of MPI's scaling results: strongly scaling, partly scaling, reverse scaling. We show these results for only T1XXL for brevity. We have verified that results on T3XXL have similar characteristics. As before, we provide exactly the same number of resources for MPI and HCMPI for fair comparison.

It is evident that for both MPI and HCMPI, *work* overshadows the *overhead* time, although HCMPI consistently shows $5\times$ smaller overhead. This is because the computation worker only ever interrupts itself to inject more work into the work-stealing deque. It never has to deal with responding to communication, something which is handled by the communication worker. For lower number of cores per node (e.g., 2 cores per node), the *work* component is higher for HCMPI compared

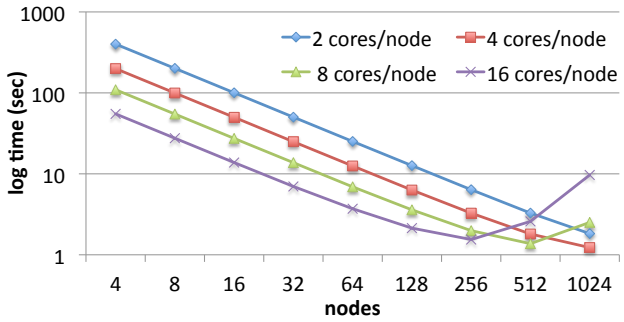


Fig. 16: Scaling of UTS for T1XXL on MPI.

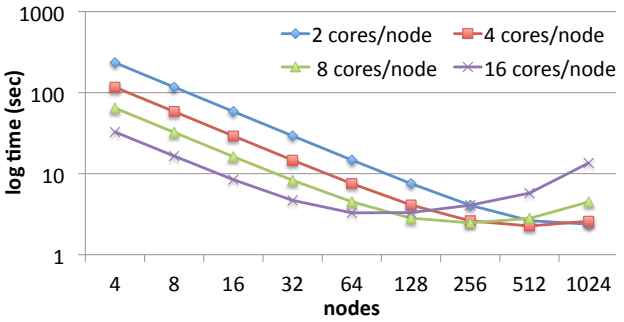


Fig. 17: Scaling of UTS for T3XXL on MPI.

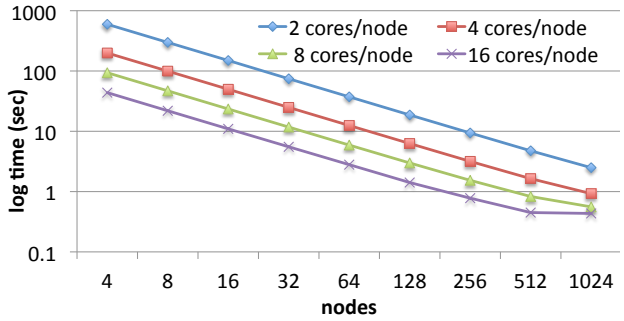


Fig. 18: Scaling of UTS for T1XXL on HCMPI.

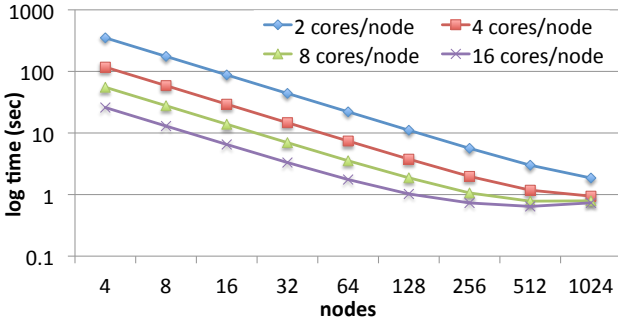
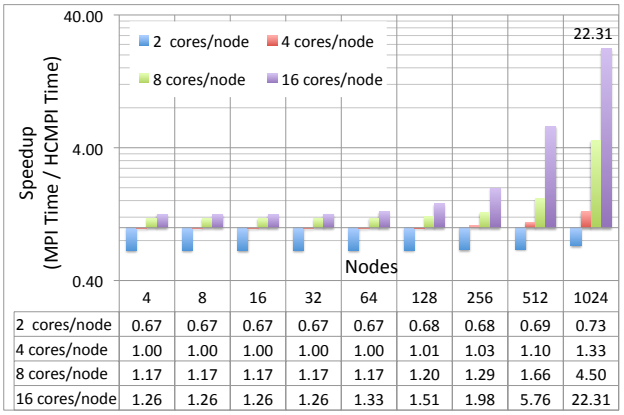
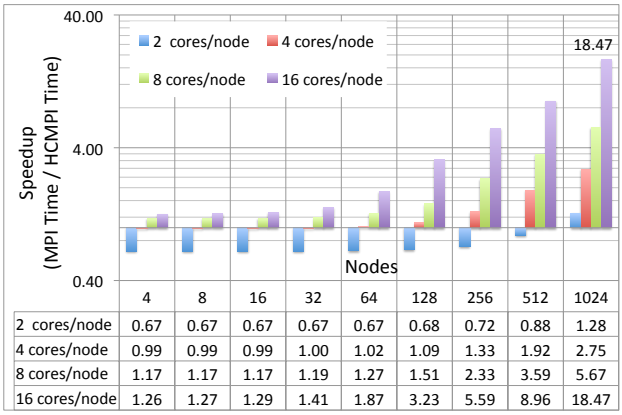


Fig. 19: Scaling of UTS for T3XXL on HCMPI.



	4	8	16	32	64	128	256	512	1024
2 cores/node	0.67	0.67	0.67	0.67	0.67	0.68	0.68	0.69	0.73
4 cores/node	1.00	1.00	1.00	1.00	1.00	1.01	1.03	1.10	1.33
8 cores/node	1.17	1.17	1.17	1.17	1.17	1.20	1.29	1.66	4.50
16 cores/node	1.26	1.26	1.26	1.26	1.33	1.51	1.98	5.76	22.31

Fig. 20: HCMPI speedup wrt MPI on UTS T1XXL



	4	8	16	32	64	128	256	512	1024
2 cores/node	0.67	0.67	0.67	0.67	0.67	0.68	0.72	0.88	1.28
4 cores/node	0.99	0.99	0.99	1.00	1.02	1.09	1.33	1.92	2.75
8 cores/node	1.17	1.17	1.17	1.19	1.27	1.51	2.33	3.59	5.67
16 cores/node	1.26	1.27	1.29	1.41	1.87	3.23	5.59	8.96	18.47

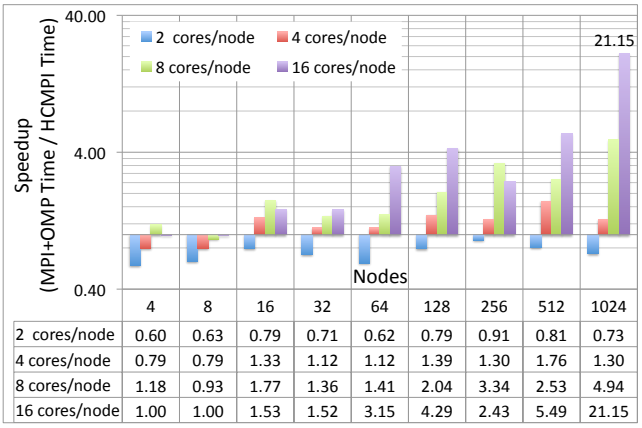
Fig. 21: HCMPI speedup wrt MPI on UTS T3XXL

to MPI which directly influences the overall running time, since HCMPI has few workers compared to MPI. For low core counts, this leads to up to 50% more work per computation worker thread. Most importantly, it is evident that for higher cores per node, the *search* component becomes the biggest bottleneck for MPI performance. For example, on 1024 nodes, when going from 8 cores to 16 cores, MPI spends $5.4\times$ more time in the *search*. In comparison, HCMPI's *search* component remains fairly stable. Consequently, HCMPI's improvement over MPI when scaling from 8 cores to 16 cores in that configuration is $22.3/4.5 \approx 5\times$. Similarly, when going from 4 to 8 cores, MPI spends $3.9\times$ more time in *search*, which is reflected in HCMPI's $4.5/1.33 \approx 3.4\times$ speedup during the same scaling when compared to MPI.

In order to understand why MPI spends more time in the *search* phase, we profiled the number of failed steal requests (see Fails column in Table III). We observed that MPI has $10.7\times$ and $5.1\times$ more failed steal requests for 1024 nodes with 16 and 8 cores per node cases respectively, which can be accounted for the bulk of extra *search* time presented before. MPI steal requests are two-sided. The thief has to send a steal request to the victim and wait for a response. Failed two-sided steals imply redundant communication, an inherent drawback of the MPI work-stealing model. On the other hand, majority of HCMPI steals are intra-node shared-memory steals where a worker thread can directly steal from another worker's deque without disturbing the victim. From these results, we conclude that HCMPI's faster stealing policy coupled with a highly responsive communication worker per node results in better computation and communication overlap and scalable performance.

1024 Nodes	MPI					HCMPI				
Cores	Time(s)	Work(s)	Overhead(s)	Search(s)	Fails	Time(s)	Work(s)	Overhead(s)	Search(s)	Fails
2	1.696	1.416	0.047	0.225	2703979	2.663	2.377	0.014	0.260	9861326
4	1.245	0.702	0.026	0.440	7869775	0.963	0.786	0.005	0.162	6279535
8	2.376	0.392	0.019	1.715	47102587	0.728	0.368	0.003	0.331	9212784
16	10.770	0.195	0.011	9.295	94754150	0.443	0.171	0.002	0.261	8835986
256 Nodes	MPI					HCMPI				
Cores	Time(s)	Work(s)	Overhead(s)	Search(s)	Fails	Time(s)	Work(s)	Overhead(s)	Search(s)	Fails
2	5.941	5.698	0.169	0.073	601384	9.641	9.511	0.053	0.076	584293
4	3.052	2.818	0.090	0.142	1603756	3.240	3.148	0.021	0.071	640242
8	1.829	1.532	0.054	0.233	2027647	1.561	1.479	0.011	0.069	562496
16	1.457	0.775	0.034	0.510	2353054	0.793	0.691	0.005	0.095	824427
64 Nodes	MPI					HCMPI				
Cores	Time(s)	Work(s)	Overhead(s)	Search(s)	Fails	Time(s)	Work(s)	Overhead(s)	Search(s)	Fails
2	23.231	22.534	0.643	0.054	33814	38.216	37.947	0.215	0.054	54509
4	11.708	11.323	0.339	0.046	127823	12.736	12.608	0.078	0.049	74264
8	6.518	6.237	0.207	0.073	456853	6.017	5.919	0.041	0.057	104471
16	3.431	3.075	0.127	0.189	203836	2.842	2.765	0.019	0.057	80501

TABLE III: UTS overhead analysis for T1XXL runs on Jaguar



	4	8	16	32	64	128	256	512	1024
2 cores/node	0.60	0.63	0.79	0.71	0.62	0.79	0.91	0.81	0.73
4 cores/node	0.79	0.79	1.33	1.12	1.12	1.39	1.30	1.76	1.30
8 cores/node	1.18	0.93	1.77	1.36	1.41	2.04	3.34	2.53	4.94
16 cores/node	1.00	1.00	1.53	1.52	3.15	4.29	2.43	5.49	21.15

Fig. 22: HCMPI Speedup wrt MPI+OpenMP on UTS T1XXL

Comparison with MPI + OpenMP: Although there is no publicly available reference implementation of UTS using MPI and OpenMP in a hybrid model, we have created one ourselves by integrating the reference MPI and OpenMP codes. In the initial implementation plan, the MPI process first ensures locally available work before starting an OpenMP parallel region to execute that work. After the threads complete execution locally, the parallel region ends and the program goes back to MPI mode to search for more global work. This naive staged approach however suffered terribly from thread idleness problems resulting in worse performance than MPI. As an improvement we increased the computation and communication overlap. In the OpenMP parallel region when threads run out of work and cannot find anything to steal, they wait at a cancelable barrier. In case more local work becomes available, the barrier gets cancelled and all threads re-enter the execution region. In our hybrid implementation, when a thread gets to the cancelable barrier, it requests for global work. So, a global steal request goes out even when some threads are busy computing. If global work arrives when the parallel region is active, the work gets folded into local work by the thread that receives it. This approach drastically improved performance over the naive implementation. We compare its performance against HCMPI on T1XXL in Fig. 22. In this experiment the hybrid code used one MPI process on every node. The number of OpenMP threads were the same as the total number of worker threads used by HCMPI, for a fair comparison.

C. Case Study: Smith-Waterman

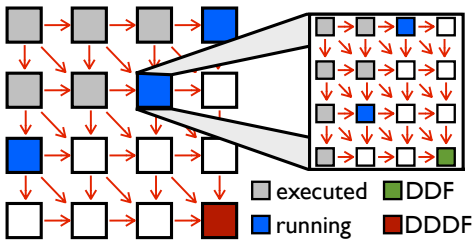


Fig. 23: Smith-Waterman dependency graph, its hierarchical tiling and execution wavefronts

We introduced a simple Smith-Waterman benchmark in section II-D and here we show a hierarchically tiled implementation of the benchmark. This implementation performs hierarchical tiling as in Figure 23. This allows us to tune granularity to minimize communication for the outer-most tiling and to minimize intra-node task creation overhead for the inner most tiling, while retaining sufficient parallelism at both levels.

An outer tile consists of a matrix of inner tiles, and three distributed DDFs. On Figure 23, we show an enlarged outer tile consisting of a matrix of inner tiles. To minimize communication, the distributed DDFs for the outer tile are the right-most column, the bottom-most row and the bottom-right element, since these are the edges of a tile visible to neighboring tiles. Similarly, an inner tile encapsulates a matrix of elements and three shared-memory DDFs to represent the intra-node visible edges of an inner tile. Given this representation of the dynamic programming matrix, we have exposed both the inter-node and intra-node wavefront parallelism through registering neighboring tiles' distributed- and shared-memory DDFs respectively.

SmithWaterman using DDDF		Nodes				
		8	16	32	64	96
Cores	2	1955.1	942.7	479.4	258.1	192.8
	4	668.9	336.3	184.1	109.5	86.6
	8	294.9	155.2	87.6	50.0	37.0
	12	192.3	102.2	57.2	32.8	24.4

TABLE IV: Scaling results for Smith-Waterman

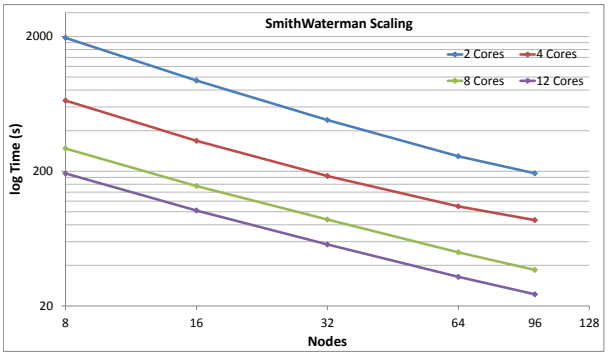


Fig. 24: Scaling results for Smith-Waterman for 8 to 96 nodes with 2 to 12 cores

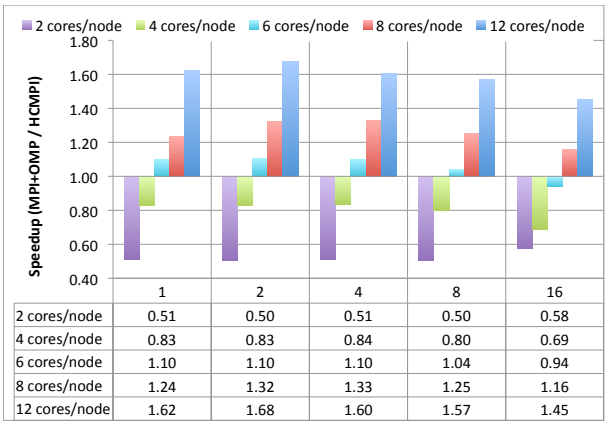
In Figure 24 and Table IV, we present a scaling study of the implementation mentioned above. Our sequences are of length 1.856M and 1.92M, giving us the dimensions for our matrix. We chose tile sizes 9280 by 9600 for outer tile sizes for a matrix of 200×200 tiles. We chose this tile size to ensure the number of wavefronts provides sufficient slackness with respect to the number of nodes. Since the number of parallel tasks at any given time is the size of an unstructured diagonal (as in figure 23), to provide enough parallelism, we need to have at least a factor of the number of nodes on most diagonals. The same logic applies to the inner tiles too, and we have chosen 290 by 300 tile sizes to have 32×32 tiles.

Using a distribution function, `DDF_HOME`, for distributed DDFs, we implemented a tiling strategy as follows. Every proper diagonal is measured in size and every contiguous chunk of that diagonal is assigned to nodes iteratively. This provides a mapping to nodes which for each node creates bands perpendicular to the wavefront and leads to less communication. This can also be interpreted as a block distribution of diagonals.

Given a fixed number of cores we observe speedups in the range $1.7 - 2$ when doubling the number of nodes until 64 nodes. This trend is hampered on the 64 node to 96 node jump because the first and last 96 diagonals do not have enough parallelism for 96 nodes, where the total number of diagonals is 399.

Given a fixed number of nodes, we observe speedups in the range 2.2-2.9 for 2 to 4 core case, since one of the workers is designated as a communication worker. The range is between 5.2-6.6 for 2 to 8 cores (for 1 to 7 computation workers), and 7.9-10.2 for 2 to 12 cores (for 1 to 11 computation workers).

On Figure 25, we contrast HCMPI DDDF with an MPI+OpenMP implementation of Smith-Waterman. The input size of strings used for this exercise are 371200 and 384000. For HCMPI DDDF, the distribution function, that produced



	1	2	4	8	16
2 cores/node	0.51	0.50	0.51	0.50	0.58
4 cores/node	0.83	0.83	0.84	0.80	0.69
6 cores/node	1.10	1.10	1.10	1.04	0.94
8 cores/node	1.24	1.32	1.33	1.25	1.16
12 cores/node	1.62	1.68	1.60	1.57	1.45

Fig. 25: HCMPI Speedup wrt MPI+OpenMP for Smith-Waterman for 1 to 16 nodes with 2 to 12 cores

the best results, is the block distribution of diagonals, as used above. MPI+OpenMP implementation produced the best results on a column distribution, which implies a cyclic distribution on the diagonals. The outer tile size 9280 by 9600 and the inner tile size 145 by 150 is favored by HCMPI DDDF, where MPI+OpenMP implementation favored outer tile size to be 5800 by 6000 and the inner tile size to be 116 by 120. Since HCMPI dedicates a worker for communication, we observe a 2× slowdown compared to MPI+OpenMP implementation when using 2 cores/node. HCMPI DDDF outperforms MPI+OpenMP beyond 6 cores per node, even though it uses one fewer computation worker. One reason for this phenomenon is the fork/join nature of MPI+OpenMP requires implicit barriers between diagonals. Every element on a diagonal can be computed in parallel and once all elements are computed, the next diagonal is started. However, HCMPI allows point-to-point synchronization through DDDFs and therefore there are no implicit barriers between diagonals. The computation wavefront may progress on an unstructured diagonal as depicted in Figure 23. Additionally, HCMPI with DDDFs allows communication/computation overlap where as the communication for the MPI+OpenMP happens after the OpenMP threads are done with the computation.

V. Related Work

While most MPI implementations, or high-performance communication systems such as Nemesis [24] and Portal [25], can identify and optimize intra-node message passing using shared-memory, the node-level core and memory architectures are mostly ignored. This limits the "MPI everywhere" model from efficiently using shared resources on a node, such as shared caches.

Studies on the PGAS model [26] have shown that threads, processes and combinations of both are needed for maximum performance as proposed by HCMPI. However, some unavoidable overheads in the PGAS model, such as locking in the thread version and network contention in the process version are overcome with HCMPI's runtime design.

In most hybrid MPI+OpenMP programming practices [27]–[31], computation is performed in OpenMP parallel regions,

and MPI operations are performed in the sequential path of the execution. OpenMP parallel threads do not participate in inter-node operations. This pattern limits the flexibility of using asynchronous MPI operations for latency hiding and computation/communication overlap. It is also difficult to fully utilize the bandwidth of multiple network interfaces that are commonly available in high-end large-scale systems.

The MPI/SMPSs [32] system closely follows the HCMPI design. MPI message passing operations are wrapped as tasks and blocking communication operations are handled by a dedicated communication thread. Yet, the HCMPI model is able to express richer functionality with "message-driven" tasks and a truly distributed data flow programming model with DDDF's.

Jégou [33] relies on a task migration model to execute program regions with non-local data. A task can spawn independent subtasks but cannot communicate or synchronize with them, in contrast to the HCMPI approach.

Cera et al. [34] evaluate MPI-2s dynamic processes as a way of supporting dynamic task parallelism in MPI. MPI-2s dynamic processes allow the dynamic creation of new MPI processes in the MPI runtime using `MPI_Comm_spawn`. While this maintains a familiar API (as HCMPI does), it also suffers from the "MPI-everywhere" limitations.

Ramaswamy et al. [35] introduce an annotated form of High Performance Fortran for extracting task and data parallelism from an application. It constructs a computation graph with a cost model for scheduling data-parallel tasks and data transfers between them in a distributed-memory machine, attempting to do automatic scheduling for the programmer. Our work provides a lower-level, but more flexible, simplified and unified programming model.

Hamidouche et al. [36] present a framework to model, evaluate, and generate hybrid MPI+OpenMP code given an input MPI code. It employs a combination of static analysis of code and dynamic analysis of the cost to estimate computation and communication costs. The best configuration (number of MPI processes and OpenMP threads) for various phases of a program are found by direct search of the shortest path on a weighted DAG where each node represents a different configuration and edges represents cost of changing from one configuration to the other. On a set of benchmarks, althought the model predicted cost vs. the actual cost are within 17% accuracy on low-end systems the numbers remain unproven on large scale systems like the Jaguar supercomputer.

Ravichandran et al. [37] present a distributed load balancing system on multi-core clusters which combines a local, work-stealing algorithm using global-local partitioned queues at the intra-node level with a dedicated thread to handle load balancing between nodes. A distributed termination detection algorithm for this system is also presented, though is not relevant to HCMPI as termination is controlled by the application in our system. This work is complementary to HCMPI, as adding the load-balancing algorithm to HCMPI would increase the flexibility of our system and allow remote execution of asynchronous tasks. However, results in this work are limited to clusters with O(10) nodes where as HCMPI is shown to scale up to O(1000) nodes.

Saraswat et al. [38] also present complementary work to ours. Their work on distributed load balancing in the X10 programming model based on "lifelines" could be used to increase the flexibility of UTS implementation with HCMPI, which depends on the application's distributed load balancing algorithm. Lifelines are links between nodes which specify which nodes can wake up other nodes with new work, after a node has put itself to sleep due to a number of failed steal attempts. By constructing lifelines using cyclic hypercubes, this work balances long paths between nodes with high degrees of edges to efficiently spread work along lifelines. HCMPI has shown very good scalability based on better computation-communication overlap and utilization of network bandwidth. Adding inter-node load balance could potentially augment the scalability and performance of HCMPI further.

Wu et al. [39] compare the performance of Hybrid MPI+OpenMP for SP and BT NAS parallel benchmarks with their corresponding MPI implementations. With the hybrid implementation they observe 21% performance gain for SP and 9% for BT. The performance is compared on Jaguar (Cray XT5) and Intrepid (BlueGene/P) systems. HCMPI distinguishes itself by offloading all communications to a dedicated communication thread so that the HCMPI computation threads can make progress, whereas in hybrid MPI+OpenMP the OpenMP threads involved in communication block. Hybrid MPI+OpenMP as presented in the paper is evaluated for only 1-25 nodes and the performance generally degrades with increasing number of cores. On the other hand, HCMPI demonstrates scalable performance improvement for 16K cores. Hybrid MPI+OpenMP also demonstrates added overhead from OpenMP on loops with computationally light loop bodies. HCMPI's work stealing runtime does not suffer from this limitation.

VI. Conclusions and Future Work

In this paper, we presented the HCMPI programming model and runtime, which unifies asynchronous task parallelism at intra-node level with MPI's message passing model at the inter-node level. With HCMPI's task parallel model, users can benefit from MPI integration with structured task parallelism and data-flow programming. Constructs such as `async`, `finish`, and `await` have been seamlessly integrated with the message model. They can create new communication tasks, wait for their completion and start execution based on message events. We have also enhanced the phaser construct for point-to-point and barrier synchronization within and across nodes.

HCMPI programs execute on a novel scalable runtime system consisting of a communication worker and a number of computation workers. For the UTS benchmark on the ORNL Jaguar machine with 1024 nodes and 16 cores/node, HCMPI performed 22.3× faster than MPI for input size T1XXL and 18.5× faster than MPI for input size T3XXL (using the best chunking and polling parameters for both HCMPI and MPI). Users can also use HCMPI to take advantage of a distributed

dataflow programming model (DDDFs) as a simple extension to the shared-memory DDF model. Scalability results for the Smith-Waterman benchmark show the practicality of this approach, which offers high programmability.

The ongoing and future work include the support for more MPI-like APIs in the HCMPI programming model, including one-sided communication operations. We are also integrating support for heterogeneous computing in HC into HCMPI, as well as exploring the integration of HC with other communication systems beyond MPI (such as GASNet and Portals).

ACKNOWLEDGMENT

The authors would like to thank all members of the Habanero group and Prof. John Mellor-Crummey at Rice University for insightful discussions and constructive feedback. We also thank all reviewers of this paper for their feedback and suggestions.

This research was supported in part by the Center for Domain-Specific Computing (NSF Expeditions in Computing Award CCF-0926127), and by the X-Stack program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR). This research was also supported by the DOE Office of Science under cooperative agreement number DE-FC02-07ER25800, and used resources of the National Center for Computational Sciences (NCCS) at the Oak Ridge National Laboratory. NCCS is supported by the DOE Office of Science under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] K. Bergman *et al.*, "ExaScale Computing Study: Technology Challenges in Achieving Exascale System," 2008.

[2] V. Sarkar, W. Harrod, and A. E. Snavely, "Software challenges in extreme scale systems," *Jour. of Phys.: Conf. Series*, vol. 180, no. 1.

[3] W. Carlson, T. El-Ghazawi, B. Numrich, and K. Yellick, "Programming in the Partitioned Global Address Space Model," 2003.

[4] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05*. New York, NY, USA: ACM, 2005, pp. 519–538.

[6] T. El-Ghazawi, W. W. Carlson, and J. M. Draper, "UPC Language Specification v1.1.1," October 2003.

[7] R. W. Numrich and J. Reid, "Co-Array Fortran for parallel programming," *ACM SIGPLAN Fortran Forum*, vol. 17, pp. 1–31, Aug. 1998.

[8] Y. Yan, S. Chatterjee, Z. Budimlic, and V. Sarkar, "Integrating mpi with asynchronous task parallelism," in *EuroMPI*, 2011, pp. 333–336.

[9] S. Chatterjee, S. Taşırlar, Z. Budimlić, V. Cavé, M. Chabbi, M. Grossman, and V. Sarkar, "Integrating asynchronous task parallelism with mpi," Department of Computer Science, Rice University, Technical Report TR12-07, Feb. 2013.

[10] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: the new adventures of old x10," in *PPPJ*, 2011, pp. 51–61.

[11] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IPDPS*, 2009, pp. 1–12.

[12] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. S. III, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *ICS*, 2008, pp. 277–288.

[13] Y. Yan, S. Chatterjee, D. A. Orozco, E. Garcia, Z. Budimlic, J. Shirako, R. S. Pavel, G. R. Gao, and V. Sarkar, "Hardware and software tradeoffs for task synchronization on manycore architectures," in *Euro-Par (2)*, 2011, pp. 112–123.

[14] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *LCPC*, 2009, pp. 172–187.

[15] S. Tasirlar and V. Sarkar, "Data-driven tasks and their implementation," in *ICPP*, 2011, pp. 652–661.

[16] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. S. III, "Phaser accumulators: A new reduction construct for dynamic parallelism," in *IPDPS*, 2009, pp. 1–12.

[17] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlić, and V. Sarkar, "Communication Optimizations for Distributed-Memory X10 Programs," in *IPDPS*, 2011, pp. 1101–1113.

[18] J. Shirako and V. Sarkar, "Hierarchical phasers for scalable synchronization and reductions in dynamic parallelism," in *IPDPS*, 2010, pp. 1–12.

[19] R. Thakur and W. Gropp, "Test suite for evaluating performance of multithreaded mpi communication," *Parallel Computing*, vol. 35, no. 12, pp. 608–617, 2009.

[20] "EPCC OpenMP Microbenchmarks," http://www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_index.html.

[21] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "Uts: An unbalanced tree search benchmark," in *LCPC*, 2006, pp. 235–250.

[22] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng, "Dynamic load balancing of unbalanced computations using message passing," in *IPDPS*, 2007, pp. 1–8.

[23] "The Unbalanced Tree Search Benchmark," http://sourceforge.net/p/uts-benchmark/.

[24] D. Buntinas, G. Mercier, and W. Gropp, "Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem," in *CCGRID*, 2006, pp. 521–530.

[25] Sandia National Laboratories, "Portals Message Passing Interface," http://www.cs.sandia.gov/Portals/.

[26] F. Blagojević, P. Hargrove, C. Iancu, and K. Yelick, "Hybrid pgas runtime support for multicore nodes," in *PGAS*, 2010, pp. 3:1–3:10.

[27] Rolf Rabenseifner, Georg Hager, Gabriele Jost, "Hybrid MPI and OpenMP Parallel Programming Tutorial, SC10, 2010," https://fs.hlrs.de/projects/rabenseifner/publ/SC2010-hybrid.html.

[28] H. Jin and R. F. V. der Wijngaart, "Performance characteristics of the multi-zone nas parallel benchmarks," *J. Parallel Distrib. Comput.*, vol. 66, no. 5, pp. 674–685, 2006.

[29] M. F. Su, I. El-Kady, D. A. Bader, and S.-Y. Lin, "A novel fdtd application featuring openmp-mpi hybrid parallelization," in *ICPP*, 2004, pp. 373–379.

[30] E. Yilmaz, R. Payli, H. Akay, and A. Ecer, "Hybrid parallelism for cfd simulations: Combining mpi with openmp," in *Parallel Computational Fluid Dynamics*, 2007, pp. 401–408.

[31] W. Pfeiffer and A. Stamatakis, "Hybrid mpi/pthreads parallelization of the raxml phylogenetics code," in *IPDPS Workshops*, 2010, pp. 1–8.

[32] V. Marjanovic, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping communication and computation by using a hybrid mpi/smpss approach," in *ICS*, 2010, pp. 5–16.

[33] Y. Jégou, "Task migration and fine grain parallelism on distributed memory architectures," in *Parallel Computing Technologies*, ser. Lecture Notes in Computer Science, 1997, vol. 1277, pp. 226–240.

[34] M. Cera *et al.*, "Challenges and issues of supporting task parallelism in MPI," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, 2010, vol. 6305, pp. 302–305.

[35] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A framework for exploiting task and data parallelism on distributed memory multicomputers," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, no. 11, pp. 1098 –1116, nov 1997.

[36] K. Hamidouche, J. Falcou, and D. Etiemble, "A framework for an automatic hybrid mpi+openmp code generation," in *SpringSim (HPC)*, 2011, pp. 48–55.

[37] K. Ravichandran, S. Lee, and S. Pande, "Work stealing for multi-core hpc clusters," in *Euro-Par (1)*, 2011, pp. 205–217.

[38] V. A. Saraswat, P. Kambadur, S. B. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *PPOPP*, 2011, pp. 201–212.

[39] X. Wu and V. E. Taylor, "Performance characteristics of hybrid mpi/openmp implementations of nas parallel benchmarks sp and bt on large-scale multicore supercomputers," *SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 56–62, 2011.