# Hybrid Parallel Programming with MPI and Unified Parallel C *

James Dinan
Dept. Comp. Sci. and Eng.
The Ohio State University
2015 Neil Avenue
Columbus, OH U.S.A.
dinan@cse.ohio-state.edu

Pavan Balaji
Math. and Comp. Sci. Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL U.S.A.
balaji@mcs.anl.gov

Ewing Lusk
Math. and Comp. Sci. Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL U.S.A.
lusk@mcs.anl.gov

P. Sadayappan
Dept. Comp. Sci. and Eng.
The Ohio State University
2015 Neil Avenue
Columbus, OH U.S.A.
saday@cse.ohio-state.edu

Rajeev Thakur
Math. and Comp. Sci. Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL U.S.A.
thakur@mcs.anl.gov

## ABSTRACT

The Message Passing Interface (MPI) is one of the most widely used programming models for parallel computing. However, the amount of memory available to an MPI process is limited by the amount of local memory within a compute node. Partitioned Global Address Space (PGAS) models such as Unified Parallel C (UPC) are growing in popularity because of their ability to provide a shared global address space that spans the memories of multiple compute nodes. However, taking advantage of UPC can require a large recoding effort for existing parallel applications.

In this paper, we explore a new hybrid parallel programming model that combines MPI and UPC. This model allows MPI programmers incremental access to a greater amount of memory, enabling memory-constrained MPI codes to process larger data sets. In addition, the hybrid model offers UPC programmers an opportunity to create static UPC groups that are connected over MPI. As we demonstrate, the use of such groups can significantly improve the scalability of locality-constrained UPC codes. This paper presents a detailed description of the hybrid model and demonstrates its effectiveness in two applications: a random access benchmark and the Barnes-Hut cosmological simulation. Experimental results indicate that the hybrid model can greatly enhance performance; using hybrid UPC groups that span two cluster nodes, RA performance increases by a factor of 1.33 and using groups that span four cluster nodes, Barnes-Hut experiences a twofold speedup at the expense of a 2% increase in code size.

*This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy under contract DE-AC02-06CH11357; by the National Science Foundation under grant #0702182; and by a resource grant from the Ohio Supercomputer Center.


*CF'10,* May 17–19, 2010, Bertinoro, Italy.


## Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

## General Terms

Design, Languages, Performance

## Keywords

MPI, UPC, PGAS, Hybrid Parallel Programming

## 1. INTRODUCTION

The Message Passing Interface (MPI) is considered to be the de facto standard for parallel programming today [11]. The flexible, feature-rich interface provided by MPI has successfully allowed many complex scientific applications to be represented and mapped efficiently to large-scale high-end computing systems. However, the amount of memory available to an MPI process is limited by each process's virtual address space; and, for a variety of scientific applications, this space is insufficient to solve emerging problems.

Many scientific applications today are written in MPI using a one-process-per-core model that partitions memory among the cores. As systems grow, memory per core remains constant or decreases. Shared memory hybrid parallel programming with MPI and OpenMP avoids partitioning of memory and, for some applications, provides access to a large enough amount of memory to simulate increasingly large problems [18]. For many other applications, however, the memory requirement grows superlinearly with problem size. In particular, the simulation of the phenomena n the nucleus of an atom via the Green's function Monte Carlo (GFMC) method has a per process memory requirement that grows as $2^A \cdot A!$ in the number of nucleons [17]. Hybridization of this MPI code with OpenMP has successfully extended it to simulate carbon-12, which requires roughly 0.5 GB memory per node. For larger atoms, however, the per-MPI-process memory requirements quickly exceed the available memory per node. Thus, a new solution is needed.

Partitioned global address space (PGAS) models such as the Unified Parallel C (UPC) [21] are relative newcomers to large-scale sci-

entific computing. However, they are gaining popularity because of their ability to provide access to a large, convenient shared global address space. This global address space can span the memory of multiple processes providing access to a large aggregate storage space rather than being restricted by each process's virtual address space. This, together with powerful locality-aware one-sided mechanisms to access the global address space, makes UPC an attractive model for space-constrained scientific codes. However, converting existing parallel programs to UPC to take advantage of its global address space is a daunting task and can require rewriting large amounts of code.

In this paper, we present a new hybrid parallel programming model that combines MPI and UPC, allowing the programmer to leverage the strengths of both models. The hybrid model provides an incremental pathway to extending existing MPI programs to utilize UPC's partitioned global address space and powerful one-sided communication features. The hybrid model also enables UPC programmers to leverage MPI for process management and to create static UPC groups connected with MPI communication links. Such groups can be applied to static, distributed, shared multi-dimensional arrays that provide the highest convenience and productivity to UPC programmers. These static groups can eliminate the complexity of manual redistribution of UPC shared data while greatly improving the scalability of UPC codes that lose performance through diminishing locality as the number of processors grows. In addition, interoperability with MPI exposes opportunities for UPC programmers to take advantage of existing parallel libraries and solvers, such as ScaLAPACK [5].

This paper presents a detailed description of the hybrid MPI+UPC parallel programming model and describes the tools and techniques needed to enable hybrid execution and mitigate the complexity of managing two parallel programming models. Using these techniques, we demonstrate the effectiveness of the model in two applications: a random-access benchmark that models the characteristics of the GFMC many-body simulation and the Barnes-Hut cosmological n-body simulation [3]. Experimental results indicate that for UPC global address spaces that span four cluster nodes, the hybrid model offers up to a *twofold* speedup for Barnes-Hut and a 25% performance boost for the random access benchmark when compared with a baseline execution on the same number of processor cores. In the case of the Barnes-Hut benchmark, the complexity cost of hybridization was a 2% in the number of lines of code. To our knowledge, this is the first such demonstration of a hybrid MPI+UPC application.

The rest of the paper is organized as follows. In Section 2 we give an overview of the MPI and UPC parallel programming models and discuss why MPI-2 one-sided messaging extensions fall short of providing the desired global address space programming model. In Section 3 we describe the hybrid MPI+UPC programming model, breaking it into three models that vary the relationship between UPC and MPI: `flat`, `nested-funneled`, and `nested-multiple`. In Section 4 we present an experimental evaluation of the hybrid model in two applications: a random-access benchmark and the Barnes-Hut n-body simulation.

# 2. OVERVIEW OF MPI AND UPC

In this section we provide an overview of the MPI and UPC parallel programming models and discuss the strengths and weaknesses of each model in order to identify the advantages of combining them to form a hybrid programming model. Overall, we find that hybridization with UPC compliments MPI codes by adding a large, asynchronous global address space and that hybridization with MPI enhances UPC by providing additional mechanisms for locality control.

## 2.1 The Message Passing Interface

The Message Passing Interface (MPI) [11] is one of the most prevalent and highly tuned parallel programming models and is available on most high-performance computing systems. The MPI-1 standard provides for two-sided messaging where communicating pairs of processes call *Send* and *Recv* to transmit a message, as well as a variety of powerful and efficient collective operations. The MPI-2 standard [12] added support for one-sided messaging that allows processes to perform remote access to exposed regions of memory. In this model, processes collectively expose windows of memory for remote access. The window may then be accessed in either active or passive target mode. Under the active target mode, the host explicitly synchronizes its window between accesses; under the passive target mode, remote processes may access the window without any explicit interaction from the host.

MPI-2's passive mode provides a model similar to UPC's global address space programming model. However, while the passive mode provides the capability to access remote memory in a one-sided manner, it is more restrictive than a global address space in terms of its consistency, coherence, and synchronization characteristics [6, 20]. These restrictions have enabled MPI-2 to be supported on a variety of systems, including those that do not provide a cache coherent memory system. However, on systems that do provide memory coherence, these same portability restrictions make it difficult for the programmer to get access to the productivity and performance gains provided by the hardware. In comparison, UPC's model assumes memory coherence and is able to support fine-grained, asynchronous communication and dynamic distributed data structures that present challenges to the MPI-2 model. For these reasons, MPI-2's model is generally considered to provide one-sided messaging rather than a global address space. The restrictions on passive mode communication can be summarized as follows:

***Access Epochs:*** The remote process must always lock a window before accessing it. The time between lock and unlock operations is referred to as the access *epoch*. All accesses, including local accesses and accesses that do not overlap, must occur within an access epoch; otherwise the program is in error.

***Ordering:*** During an access epoch, puts and gets to a window may be reordered by the runtime system. Thus, if a location is written to within an epoch, it cannot be read from or written to again within the same epoch. This restriction makes algorithms that perform read-modify-write operations extremely difficult to implement.

***Concurrency:*** All accesses to a window, whether local or remote, must occur within a locked access epoch. Hence, concurrency may be greatly impacted if the number of windows is low. Because MPI-2's locks are associated with window objects, the programmer may be compelled to create multiple windows in order to achieve finer-grained locking.

***Window Creation:*** Window creation is a collective operation. Thus, the creation of multiple windows to increase concurrency must be done collectively. Hence, handling dynamic allocation of elements in distributed data structures (e.g., trees) may require the user to create a system for managing preallocated buffers.

***Shared Pointers:*** MPI window objects cannot be transferred between processes. Thus, creation of distributed linked data structures requires extra bookkeeping in order to create an out-of-band system for managing portable shared pointers.

## 2.2 Unified Parallel C

UPC is a extension of the C programming language that adds support for parallel programming with distributed, shared data. UPC is supported on both shared- and distributed-memory systems and presents the programmer with a logically partitioned global address space that is physically distributed across available memory domains. Under UPC, every shared data element has affinity to a distinct processor, and UPC exposes this data locality information to the programmer so that it can be leveraged to enhance performance. Shared data can be accessed through built-in language-level support as well as through explicit one-sided communication operations. Regardless of location, all data in the global address space can be accessed without explicit help from the data's owner.

A distinct advantage of UPC over libraries such as MPI is that, because of its language extensions, UPC can offer a tunable approach to performance. The programmer may start with a sequential program and convert it to a simple, shared-memory implementation. From this implementation, the programmer can then incrementally enhance the performance by tuning data locality and array distributions and by relaxing the memory model. For performance-critical sections of the code, the programmer can delve more deeply through explicit memory management and one-sided communication.

In spite of these advantages, UPC does not provide process groups, and UPC's distributions do not provide the flexibility needed to allocate distributed shared arrays on a subset of processors. Thread teams have been proposed as a UPC extension. However, it is unclear how dynamically created teams can be applied to statically declared distributed shared multidimensional arrays, which offer the highest convenience to UPC programmers. As we discuss in Section 3, the hybrid MPI+UPC model offers a solution that allows the programmer to replicate static shared arrays over hybrid groups.

# 3. MPI + UPC HYBRID MODEL

The hybrid MPI+UPC programming model combines MPI and UPC in the same program, allowing the programmer to take advantage of MPI's locality control and UPC's global address space. Because UPC is an extension to the C programming language and MPI is a library, a hybrid program is simply a UPC program with calls to the MPI library. This program is compiled with the UPC compiler and linked with the MPI libraries. Many existing scientific applications are written in MPI using Fortran. These programs can still benefit from hybridization by linking with an external UPC library that provides access to data stored in a global address space.

In the UPC terminology, a single unit of execution is referred to as a UPC thread. Each thread in a UPC execution is given the constants MYTHREAD and THREADS to identify itself in the computation. This terminology is meant to emphasize UPC's goal of providing shared-memory-like programming. In practice, however, most UPC distributions implement UPC threads as operating-system-level processes. In this work we focus on this model, but the techniques and hybrid programming model can also be applied to UPC implementations that use threads. Thus, in our discussion we assume that processes in a UPC application all share a common global address space but have distinct private virtual address spaces and instances of the MPI library. For this reason, unlike with hybrid MPI+OpenMP model, sharing a single MPI rank between multiple UPC processes is not easy. Yet such sharing is desirable because we want UPC global address spaces to span multiple cluster nodes and MPI does not support sharing a rank across cluster nodes.

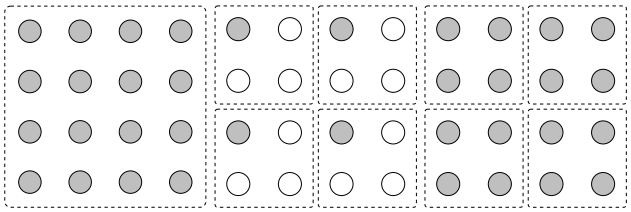
(a) Flat (b) Nested-Funneled (c) Nested-Multiple

**Figure 1: MPI+UPC hybrid execution models; gray circles represent hybrid MPI+UPC processes, and white circles represent UPC-only processes.**

We define the hybrid MPI+UPC programming model in terms of submodels that vary the level of nesting and number of instances of both models. A representation of these models is shown in Figure 1. The first model, referred to as the `flat` model, provides a nonnested common MPI and UPC execution where each process is a part of both the MPI and the UPC execution. Thus, each process has both a unique MPI rank and a UPC thread identifier.

The second and third models show two `nested` modes where multiple UPC executions have been launched in the context of a single, outer MPI environment. Each UPC process can communicate via UPC only within its group. Intergroup communication is performed by using MPI. The `nested-funneled` model provides an operational mode similar to the funneled mode in hybrid MPI+threads. That is, only the master process per group gets an MPI rank and can make MPI calls. The `nested-multiple` model, on the other hand, provides a mode where every UPC process gets its own MPI rank and can make MPI calls independently.

When a hybrid MPI+UPC program is executed, the execution is parameterized by a set of ranks and a set of group sizes. The hybrid model inherits the UPC constants `THREADS` and `MYTHREAD`, which specify the number of UPC threads (i.e., processes) within a UPC execution and an individual thread's rank. Similarly, each process in the outer MPI execution has a rank within the MPI `WORLD` communicator and is provided with the size of the MPI execution. In the `nested` model, where multiple UPC instances are launched as part of the same outer MPI environment, the execution is further parameterized by a group rank and the number of groups.

Thus, from the point of view of a single computational entity in a hybrid computation, the structure of the hybrid execution can be fully described by the group, UPC THREAD ID, and MPI rank,

$$ID = \langle id_{group}, id_{UPC}, id_{MPI} \rangle,$$

and by the number of UPC groups, UPC group size (we assume a model where all UPC groups are the same size), and MPI communicator size,

$$N = \langle n_{group}, n_{UPC}, n_{MPI} \rangle.$$

These two sets of parameters fully describe all hybrid MPI and UPC combinations. However, this model may be unnecessarily complex for some codes. In the next two subsections, we describe two restrictions on this model, the `flat` and `nested-funneled` models, which offer lower complexity when the full model is not needed. We also describe the full `nested-multiple` model and provide techniques to mitigate complexity and improve programmability. For each model, we give an example hybrid dot product code. This simple code is not intended to motivate the hybrid model because efficient MPI- and UPC-only implementa-

```
#include <upc.h>
#include <mpi.h>

#define N 100*THREADS
shared double v1[N], v2[N];

int main(int argc, char **argv) {
  int i, rank, size;
  double sum = 0.0, dotp;
  MPI_Comm hybrid_comm;

  MPI_Init(&argc, &argv);
  MPI_Comm_split(MPI_COMM_WORLD, 0,
    MYTHREAD, &hybrid_comm);
  MPI_Comm_rank(hybrid_comm, &rank);
  MPI_Comm_size(hybrid_comm, &size);

  upc_forall(i = 0; i < N; i++; i)
    sum += v1[i]*v2[i];

  MPI_Reduce(&sum, &dotp, 1, MPI_DOUBLE,
    MPI_SUM, 0, hybrid_comm);
  if (rank == 0) printf("Dot = %f\n", dotp);

  MPI_Finalize();
  return 0;
}
```

**Figure 2: Hybrid dot product example in the `flat` model.**

tions are possible. Instead, we wish to more concretely describe the model through these examples.

## 3.1 Flat Model

The `flat` model is the most straightforward hybrid MPI+UPC model. This model, shown in Figure 1(a), restricts the full model by fixing the number of groups at one. Thus, the model is composed of conjoined UPC and MPI executions where all processes participate in both UPC and MPI communication. In this model, $n_{group}$ is 1, and thus the group rank and size can be ignored. Also, $n_{UPC} = n_{MPI}$, allowing the programmer to eliminate either the UPC or the MPI parameter. Thus, execution under this model can be parameterized by $ID = \langle id_{MPI} \rangle$ and $N = \langle n_{MPI} \rangle$.

An example `flat`-style hybrid program is shown in Figure 2. This program computes the dot product of two distributed shared vectors by first finding the partial dot product of all the local elements and then using an MPI reduction to find the sum the partial sums.

In this program, all UPC threads participate in MPI communication and must initialize MPI. There is no guarantee that MPI will assign ranks equal to the UPC thread IDs because it is unaware of UPC. In order to renumber the MPI space so that MPI and UPC ranks are equal, a new communicator is formed by calling `MPI_Comm_split()`. In the call, all processes provide the same color to indicate they all wish to join the same communicator and provide their UPC thread ID as the key. The key values are then sorted by MPI, and ranks are distributed in order of the keys provided. Since each UPC thread provides MYTHREAD as the key, its rank in the new MPI communicator will be equal to MYTHREAD.

## 3.2 Nested Model

The `nested` model, shown in Figure 1(b) and 1(c), can be viewed in two ways. For MPI programmers, this model reflects an outer MPI execution that contains several UPC executions, each with its own global address space. For UPC programmers, this model can be viewed as multiple instances of their UPC program connected using an outer layer of MPI. We refer to each UPC execution as a UPC group and parameterize the `nested` execution using the group rank and the number of groups as well as the number of UPC threads per group (we assume groups are of uniform size) and the number of MPI ranks per group.

### 3.2.1 Nested-Funneled Model

When the number of MPI ranks per group is restricted to 1, we call this the `nested-funneled` model. In this model, only one process per UPC group can participate in MPI communication. The reason is UPC threads are generally implemented as distinct processes that cannot share an MPI rank. Thus, in the `nested-funneled` model, the number of MPI ranks must equal the number of groups. This restriction allows the MPI rank of a group to be used as its group ID and allows the total number of MPI ranks to determine the number of groups. Thus, this model is parameterized by the UPC and MPI ranks and process counts: $ID = \langle id_{UPC}, id_{MPI} \rangle$ and $N = \langle n_{UPC}, n_{MPI} \rangle$.

In Figure 3 we show the dot product example written for the `nested-funneled` hybrid model. In this model, only one member of each UPC group (`MYTHREAD == 0`) initializes MPI and is able to make MPI calls. Once this master thread has initialized MPI, it does not need to remap MPI's ranks, as in the `flat` model, because the ranks will be used as the UPC group IDs. However, the MPI rank and size must be made available to all UPC threads in the group because they also indicate the group rank and number of groups. Therefore, `rank` and `size` are stored in shared variables accessible by all threads.

The `nested` hybrid model has two explicit levels of parallelism. Hence, the work must be partitioned twice, once across groups and again within groups. The partitioning across groups is done in blocks with block size $B$. Partitioning within each UPC group is done by using the `upc_forall` construct, which distributes iterations of the loop according to the affinity expression, `&v1[i]`. UPC will assign each iteration to the thread that has affinity to the corresponding element of `v1`.

Once the partial sums are computed for each thread, the partial sum for the group is computed by performing a `upc_all_reduceD` across each UPC group. This operation has a synchronized entry and exit to ensure that all values are ready when the reduction starts and that all UPC communication is completed before performing the next step. Finally, the master thread performs an MPI reduction to find the global sum. Only the master thread finalizes MPI before the program exits.

### 3.2.2 Nested-Multiple Model

`Nested-multiple` is the most powerful because it allows MPI to span all processes in all groups. However, this added flexibility comes with greater complexity. Because the number of UPC and MPI processes and groups may all differ, this model requires the full parameterization: $ID = \langle id_{group}, id_{UPC}, id_{MPI} \rangle$ and $N = \langle n_{group}, n_{UPC}, n_{MPI} \rangle$.

In Figure 4 we show the dot product example written for the `nested-multiple` execution model. In the `nested-multiple` model multiple UPC groups are launched, and all processes in each group initialize MPI. There is no guarantee that MPI will be able to assign ranks that are contiguous within a UPC group because it is unaware of UPC. Thus, each group has to identify its group ID using a separate mechanism. For example, all MPI processes with UPC thread ID *zero* can exchange their MPI ranks with each other, sort them, and use this ordering as its UPC group ID. However, if

```
#include <upc.h>
#include <mpi.h>

#define N 100*THREADS
shared double v1[N], v2[N];
shared double our_sum = 0.0;
shared double my_sum[THREADS];
shared int me, np;

int main(int argc, char **argv) {
  int i, B;
  double dotp;

  if (MYTHREAD == 0) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,(int*)&me);
    MPI_Comm_size(MPI_COMM_WORLD,(int*)&np);
  }
  upc_barrier;

  B = N/np;
  my_sum[MYTHREAD] = 0.0;
  upc_forall(i=me*B;i<(me+1)*B;i++; &v1[i])
    my_sum[MYTHREAD] += v1[i]*v2[i];

  upc_all_reduceD(&our_sum,&my_sum[MYTHREAD],
    UPC_ADD, 1, 0, NULL,
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC)

  if (MYTHREAD == 0) {
    MPI_Reduce(&our_sum, &dotp, 1, MPI_DOUBLE,
        MPI_SUM, 0, MPI_COMM_WORLD);
    if (me == 0) printf("Dot = %f\n", dotp);
    MPI_Finalize();
  }

  return 0;
}
```

**Figure 3:** `Nested-funneled` **dot product.**

```
#include <upc.h>
#include <mpi.h>

#define N 100*THREADS
shared double v1[N], v2[N];
shared int r0;

int main(int argc, char **argv) {
  int i, me, np;
  double sum = 0.0, dotp;
  MPI_Comm hybrid_comm

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  if (MYTHREAD == 0) r0 = me;
  upc_barrier;
  MPI_Comm_split(MPI_COMM_WORLD, 0,
    r0*THREADS+MYTHREAD, &hybrid_comm);
  MPI_Comm_rank(hybrid_comm, &me);
  MPI_Comm_size(hybrid_comm, &np);

  int B = N/(np/THREADS); // Block size
  upc_forall(i=me*B;i<(me+1)*B;i++; &v1[i])
    sum += v1[i]*v2[i];

  MPI_Reduce(&sum, &dotp, 1, MPI_DOUBLE,
    MPI_SUM, 0, hybrid_comm);

  if (me == 0) printf("Dot = %f\n", dotp);

  MPI_Finalize();
  return 0;
}
```

**Figure 4:** `Nested-multiple` **dot product.**

the MPI ranks are renumbered to be contiguous across groups and given that we have restricted groups to be of uniform size, we can find the group ID by dividing the MPI rank of any process by the size of a group $id_{group} = id_{MPI}/THREADS$. Likewise, since all processes are part of the outer MPI environment, the number of groups can be found by dividing the number of MPI ranks by the size of a group, $n_{group} = n_{MPI}/n_{UPC}$.

This renumbering of MPI ranks can be accomplished by exchanging $r_0$ the MPI rank of thread 0 in every group and calling *MPI_Comm_split* with the same color and $key = r_0{*}THREADS + MYTHREAD$. The result is a new communicator where the MPI ranks have been assigned contiguously across UPC groups.

Once renumbering is complete, the work is partitioned into blocks of indices assigned to each group and again into individual iterations assigned the each UPC thread within a group. In contrast to the two-step reduction required by the `nested-funneled` model, in the `nested-multiple` model where all threads participate in the outer level of MPI parallelism the partial sums can be reduced to the global sum directly through a collective call to `MPI_Reduce`.

## 3.3 Launching Hybrid Computations

Launching a hybrid MPI+UPC application requires nesting one or more UPC launches within an MPI launch. MPI and UPC both require bootstrap mechanisms that spawn processes either through an existing remote shell service or through specialized servers and pass execution information to the new processes through the environment or command line arguments. This information informs the new process of how to join the computation, often by providing the location of a bootstrap server that can be reached over an existing communication channel such as TCP/IP. In this section we describe tools and techniques that can be used to launch hybrid computations. The complexity involved in launching hybrid jobs can easily be hidden from the user via a wrapper that automates this process.

### *3.3.1 Nested-Funneled Launching*

The `nested-funneled` model requires an MPI launcher that supports MPMD launching in order to provide different parameters to each UPC group. Below is an example launch where each MPMD task provided to `mpiexec` is separated by a ":".

```
$ mpiexec \
    -env HOSTS=hosts.1 upcrun -n N myapp \
  : -env HOSTS=hosts.2 upcrun -n N myapp \
  : ...
```

In this example several MPI tasks are launched, each of which executes upcrun. Each instance of upcrun is supplied with a different value for the environment variable `HOSTS`, which informs `upcrun` which hosts it should launch the UPC group on. Once UPC has bootstrapped and brought each group online, a single UPC thread from each group will call `MPI_Init` to start MPI and request an MPI rank.

### 3.3.2 Flat and Nested-Multiple Launching

When `flat` and `nested-multiple` hybrid applications are launched, each task (i.e., UPC group) that the MPI launcher starts will request multiple ranks. In contrast, in the `nested-funneled` case each task requests only a single rank. We have extended the Hydra process manager that is included with the MPICH2 MPI distribution [1] to allow the user to specify how many ranks will be assigned to each task that is launched. This specification is accomplished through the `-ranks-per-proc` flag.

Using the Hydra process manager, a `flat` hybrid program is launched as follows:

```
$ mpiexec --ranks-per-proc=N \
   upcrun -n N myapp
```

In this case, a single task is launched that will in turn launch an $N$-process UPC execution. Each UPC thread will then initialize MPI, requesting $N$ total ranks from the single task that `mpiexec` has launched. Thus, `-ranks-per-proc` informs the MPI process manager of this change in the MPI bootstrapping model.

A `nested-multiple` hybrid program is launched as follows:

```
$ mpiexec --ranks-per-proc=N \
   -env HOSTS=hosts.1 upcrun -n N myapp \
 : -env HOSTS=hosts.2 upcrun -n N myapp \
 : ...
```

In this case, multiple MPI tasks are launched, and each will spawn an $N$ process UPC execution. The `-ranks-per-proc` flag is used to inform the MPI process manager that each task it launched will require $N$ ranks.

# 4. EXPERIMENTAL EVALUATION

We evaluate the hybrid programming model using two benchmarks: a random access benchmark and a hybrid implementation of the Barnes-Hut cosmological n-body simulation. These applications were chosen because their baseline UPC implementations exhibit poor performance as the number of processors is increased, due to a corresponding decrease in locality. In addition, both hybrid MPI+UPC benchmarks demonstrate the use of distributed, shared data structures that are challenging to implement with MPI alone: a distributed shared array that supports fine-grained, one-sided access and a distributed shared tree.

## 4.1 Random-Access Benchmark

The random-access benchmark models an application where a large, distributed shared array is accessed by all processors with an irregular access pattern. Replicating the array on all processors is not possible because of memory limitations, and exchanging information via a regular communication pattern is not feasible because of the irregular access pattern. This benchmark is motivated by Green's function Monte Carlo codes where irregular accesses are performed on large shared arrays and information is accumulated locally. Periodically this local information is contributed to the global solution [17].

The random access benchmark uses a static UPC distributed shared array of the form:

```
shared double data[n];
```

The array is cyclically distributed with the default block size of 1 across all threads. Once the array has been initialized, every process performs a fixed number of accesses to random elements in the array. For each access, they perform 1,000 floating-point operations of computation. When run in full UPC mode, shown in Figure 5(a), a single array of $n$ elements is distributed across all UPC threads. Thus $n/THREADS$ elements are local to each thread. When run in hybrid mode, the benchmark executes in the `nested-multiple` model, as shown in Figure 5(b). In this model, multiple UPC groups will be launched, and each UPC group will have a full replica of the array that is distributed across only the UPC threads in the group. For groups of size $G$, therefore, $n/G$ elements will be local to each thread. This benchmark demonstrates replication of shared arrays through hybridization and effectively trades space for locality.

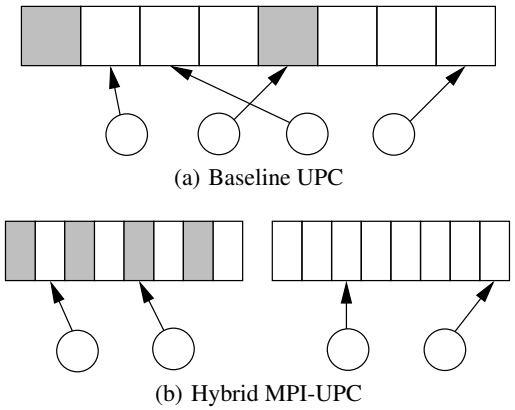


**Figure 5: Random-access benchmark snapshot on 4 processors. In *(a)* a single copy of the array is distributed across all 4 processors; in *(b)* two groups of size two each have a full replica of the array. Elements with affinity to processor 0 are shaded; in *(b)* twice as many elements are local to processor 0 than in *(a)*.**

## 4.2 Hybrid N-Body Simulation

Barnes-Hut is an $n$-body cosmological simulation algorithm [3]. This algorithm simulates the pairwise gravitational interactions of $n$ bodies distributed through a 3D region of space. Each body has a position, mass, and velocity associated with it; the algorithm simulates the motion and interaction of the bodies given their initial conditions. Barnes-Hut improves on the naive $O(n^2)$ algorithm by summarizing the interaction of distant bodies as an interaction with the center of mass of their region of space. Thus, Barnes-Hut has a computational complexity of $O(n\ log\ n)$.

In Algorithm 1 we present the hybrid MPI+UPC Barnes-Hut algorithm written for the `nested- funneled` execution model. Barnes-Hut iterates the simulation for $t_{max}$ time steps. In each time step a new oct-tree is created to represent the decomposition of the 3D region of space under simulation, and the bodies are loaded into the tree. Each node in the oct-tree represents a volume of 3D space. When a node is split, it is split in half along each dimension, resulting in $2^3 = 8$ new children.

Once the tree has been built, a bottom-up traversal is done to summarize the center of mass for each subtree. Next, the bodies are evenly partitioned across all processors using a space-filling curve. When partitioning is complete, each process computes the interactions of each of its bodies to find its state in the next time step. Once all processors finish the force computation, the list of bodies is updated to reflect their new states. Next, a groupwise gather is done to assemble the new bodies into `our_bodies`. This groupwise gather is characteristic of the `nested-funneled` model because each group has a single MPI rank making for a multilevel

**Algorithm 1** Hybrid `nested-funneled` Barnes-Hut algorithm.

```
for i = 1 to t_max do
  our_bodies ← ∅
  T' ← T
  T ← Octtree_Create()
  my_bodies ← partition(T', bodies, group_id, thread_id)

  for all b ∈ my_bodies do
    T.insert(b)
  end for
  summarize_subtrees(T)
  for all b ∈ my_bodies do
    compute_force(b, T)
  end for
  for all b ∈ my_bodies do
    advance(b)
  end for

  our_bodies = our_bodies ∪ my_bodies
  upc_barrier
  if MYTHREAD = 0 then
    MPI_Allgather(our_bodies, bodies)
  end if
end for
```

parallel programming model. Finally, the masters of each group perform an MPI all gather to gather the updated list of bodies on each UPC group.

The total number of source lines of code for the original UPC implementation of Barnes-Hut was 2,454 and the number of lines of code after hybridization with MPI was 2,505. Thus, hybridization resulted in 51 additional lines of code for an overall code size increase of 2%. In the `nested-funneled` example in Figure 3 we saw that the code required to manage both models is modest. New code in Barnes-Hut comes primarily from enhancing work (body) distribution to be multi-level parallel and from gathering of results.

## 4.3 Performance Evaluation

Experiments were conducted an 877-node IBM 1350 cluster located at the Ohio Supercomputing Center. This cluster is configured with two dual-core 2.6G Hz AMD Opteron processors and 8 GB RAM in each node; the interconnect is 10 GBps Infiniband.

For our evaluation we used the Intrepid GCCUPC v2.3.4.6 compiler with the Berkeley UPC v2.8.0 runtime. The runtime was configured to use the high-performance Infiniband interconnect with the SSH bootstrap. Using the SSH bootstrap instead of the default MPI bootstrap allows us more flexibility in the interaction between UPC and MPI. For our MPI we used MVAPICH-2 v1.2 with the Hydra process manager, which provides MPMD launching support. MPI and the Berkeley UPC runtime were compiled using the Intel C compiler and hybrid applications were compiled using GCCUPC and linked with the MPI and Berkeley UPC libraries.

For each benchmark we show the performance for a baseline UPC version and hybrid MPI+UPC versions with groups of 4, 8, and 16 cores. Groups of 4 cores span a single node and represent an ideal performance because all data in the UPC global address space is local. On this system, 4-core groups can also be achieved by using the hybrid MPI with OpenMP or threads models. However, larger groups that span multiple nodes are not possible in these models. In practice, the lower bound on group size will be dictated by an application's memory requirements and may require groups that span multiple nodes.

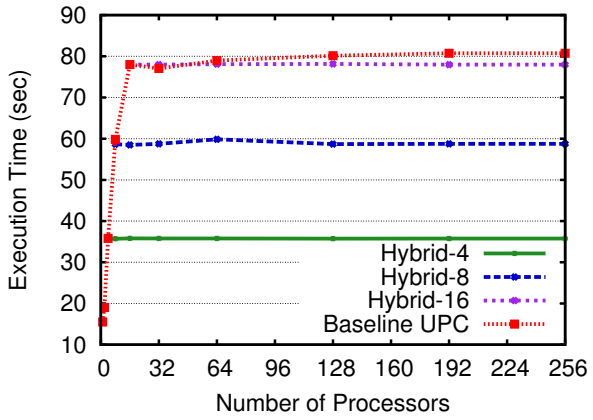


**Figure 6: Random-access benchmark execution time for baseline UPC and hybrid implementations.**

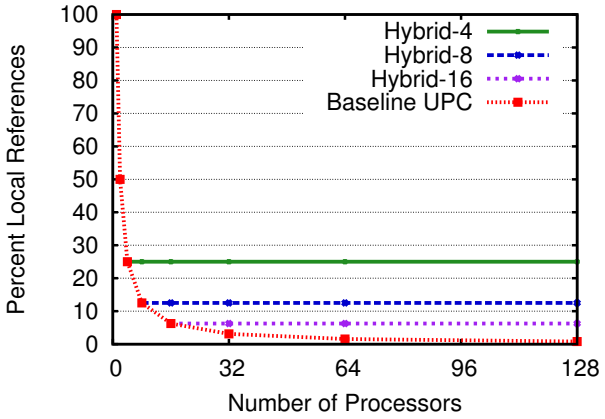


**Figure 7: Random-access benchmark percentage of local references vs number of processors.**

### 4.3.1 *Random-Access Benchmark Performance*

A weak scaling experiment was run with the random-access benchmark to show the effect of data locality on performance. Results in Figure 6 show the time required for each process to perform 1,000,000 random accesses to a distributed shared array with 1,000,000 elements. Timings are reported for the baseline UPC implementation and MPI+UPC implementations with group size 4, 8, and 16. The ideal performance for this experiment is a flat line because the work per processor is constant.

For the baseline UPC implementation, the time rapidly increases, indicating that the average latency of each array access is increasing proportionally to the number of processors. For the hybrid implementations, however, the time remains flat indicating that the average latency remains fixed regardless of the processor count.

This trend is explained by the data shown in Figure 7. In this graph, we show the percentage of all accesses to the shared array that access a local element of the array. Because the element accessed is chosen with uniform randomness, this data also shows the percentage of the array that is local to any processor. For the baseline UPC implementation, the percentage of local data decreases proportional to the number of processes as $n/THREADS$. In

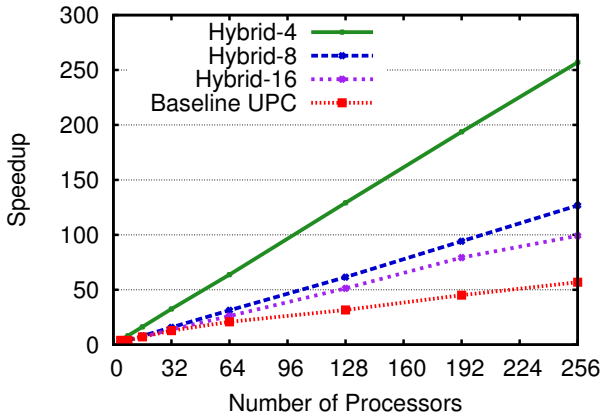


**Figure 8: Barnes-Hut force calculation performance for baseline UPC and hybrid implementations.**

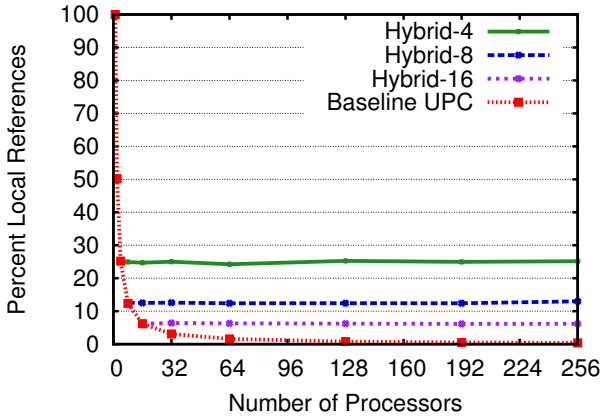


**Figure 9: Barnes-Hut force calculation locality, shown as the percentage of local data accesses.**

comparison, for a hybrid implementation with group size $G$ the percentage of local data decreases in proportion with the group size as $n/G$ rather than with the number of processors.

If we express the average latency in terms of the latency $l_l$ to access a local element, the latency $l_r$ to access a remote element, and $P(l)$ the probability that a randomly selected element will be local, we have $l_{avg} = l_l \cdot P(l) + l_r \cdot (1 - P(l)) = (l_l - l_r) \cdot P(l) + l_r$. Thus, as the number of processors is increased and $P(l)$ approaches 0, the average latency approaches $l_r$, which is reflected in the data for the baseline UPC implementation on large processor counts.

### 4.3.2 *Barnes-Hut Performance*

Strong scaling experiments were conducted with the Barnes-Hut force computation kernel. The results are reported in Figure 8 are for a 150,000-body system. We show the speedup for the baseline UPC implementation as well as for hybrid implementations with UPC group sizes of 4, 8, and 16. In Figure 9 we show the percentage of node references in the shared oct-tree that were local as the average over all processes.

From this data, we can see that the baseline UPC implementation scales poorly because the number of nonlocal references is increasing proportional to the number of UPC threads. In comparison, the hybrid schemes offer better performance because the tree has been replicated on each UPC group, giving a substantial increase in the percentage of local data references over the baseline implementation. The hybrid scheme with group size of 4 achieves almost linear scaling because this schemes fixes the group size at the size of an SMP node and all data is local. When two nodes are combined to form a group of size 8 and provide access to twice as much memory, a twofold performance increase over the baseline is achieved.

## 5. RELATED WORK

Hybrid parallel programming is the practice of combining multiple parallel programming models within a single application. Hybrid programming with MPI as an outer level of parallelism and OpenMP or threads as an inner, nested level of parallelism is becoming increasingly popular and has been extensively explored [2, 10, 18]. In this context, hybridization has been shown to help provide more efficient use of memory and improve the performance of many applications by leveraging shared memory to localize communication and improving load balance through malleable parallelism [9].

The Global Arrays (GA) toolkit for distributed, shared multidimensional arrays [14] is a successful PGAS model that is compatible with MPI for hybrid programming and provides a flat hybrid model. However, GA focuses specifically on shared array data and does not provide support for arbitrary shared, linked data structures. The Aggregate Remote Memory Copy Interface (ARMCI) [13] defines a low-level, one-sided communication library that provides the PGAS substrate for GA. ARMCI is fully interoperable with MPI and forms a hybrid MPI+PGAS programming model. However, ARMCI is very low level and does not provide the same convenience, programmability, and completeness (e.g., shared pointers, collectives) as UPC.

Some work has been done by the Berkeley UPC group to add support for MPI compatibility to UPC [4]. This mode of hybridization is targeted primarily at supporting the `flat` hybrid model for interoperability with existing MPI parallel libraries. To our knowledge, however, this paper presents the first exploration of a nested MPI+UPC hybrid programming model. The idea of using groups to improve the locality of PGAS applications has been explored in the context of Global Arrays groups and mirrored arrays [15, 16]. In addition, processor teams have been proposed as an extension to the UPC standard. However, these teams would not apply to static shared multidimensional arrays and would require such arrays to be dynamically allocated, complicating indexing. Hybrid MPI+UPC groups offer an alternative approach to dynamic processor teams by replicating static shared structures across groups.

Chapel [7], X10 [8], and Fortress [19] are new, high-productivity parallel programming languages with features that offer a union of benefits from MPI's explicit control over data locality and UPC's convenient global shared view of data. For existing programs, there is a significant barrier to adoption of these new models because of the cost of reimplementation. Hybrid MPI+UPC offers an alternative incremental pathway adopting these features in existing programs. In addition it will offer a testbed for evaluation of the benefits and trade-offs of partitioned-view versus global-view parallel models with respect to performance and productivity through incremental modification of existing applications. Thus, it can serve as a testing ground for developing insights that will better facilitate the development and implementation of new languages offering features for locality control as well as shared-global views.

The MPI Forum is working toward the next MPI standard, MPI-3, and is actively investigating enhancements to improve the flexibility and usability of MPI's one-sided communication model [20] as well as improving interoperability for hybrid programming models, including UPC. Hybrid MPI+UPC can offer a different and

more convenient model than MPI with one-sided communication because it is able to leverage UPC's high-level programming interface and tunable performance model in comparison with MPI's library approach that requires explicit communication management.

# 6. CONCLUDING REMARKS

In this paper, we have explored the hybrid programming model formed by combining MPI and UPC. This model offers an incremental pathway that allows existing applications to take advantage of MPI's locality control and UPC's global address space. In addition, it can serve as a testbed for developing new programming models that aim to combine these features. For memory-constrained MPI codes, the hybrid model enables the processing of larger problems by aggregating the memory of several nodes into a single, shared global address space. For locality-constrained UPC codes, the hybrid model can improve locality through the creation of UPC groups that are connected with MPI.

We evaluated this new model on two benchmarks, a random access benchmark and the Barnes-Hut $n$-body simulation. Compared against a baseline execution on 256 cores, we found that, for groups that span two cluster nodes, the hybrid random access benchmark yields a 25% improvement in execution time and hybrid Barnes-Hut experiences a *twofold* speedup. In the case of Barnes-Hut the cost of hybridization was a 2% increase in code size.

# 7. REFERENCES


[1] MPICH2. `http://www.mcs.anl.gov/research/projects/mpich2/`, December 2009.

[2] Eduard Ayguade, Marc Gonzalez, Xavier Martorell, and Gabriele Jost. Employing nested OpenMP for the parallelization of multi-zone computational fluid dynamics applications. *J. Parallel Distrib. Comput.*, 66(5):686–697, 2006.

[3] Joshua E. Barnes and Piet Hut. A hierarchical o(n log n) force calculation algorithm. *Nature*, 324:446–449, 1986.

[4] Berkeley UPC. Berkeley UPC user's guide version 2.8.0, 2009.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. SIAM, Philadelphia, PA, 1997.

[6] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. In *2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing (SHPSEC)*, pages 91–99, 2003.

[7] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *Intl. J. High Performance Computing Applications (IJHPCA)*, 21(3):291–312, 2007.

[8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Intl. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 519–538. ACM SIGPLAN, 2005.

[9] Julita Corbalán, Alejandro Duran, and Jesús Labarta. Dynamic load balancing of MPI+OpenMP applications. In *Intl. Conf. on Parallel Processing (ICPP)*, 2004.

[10] Haoqiang Jin and Rob F. Van der Wijngaart. Performance characteristics of the multi-zone NAS parallel benchmarks. In *18th Intl. Parallel and Distributed Processing Symp. (IPDPS)*. IEEE, 2004.

[11] MPI Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, Knoxville, 1994.

[12] MPI Forum. MPI-2: Extensions to the message-passing interface. Technical report, University of Tennessee, Knoxville, 1996.

[13] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586, 1999.

[14] Jarek Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A portable "shared-memory" programming model for distributed memory computers. In *Supercomputing (SC) '94*, pages 340–349, 1994.

[15] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and performance of the Global Arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, 2006.

[16] Bruce Palmer, Jarek Nieplocha, and Edoardo Apra. Shared memory mirroring for reducing communication overhead on commodity networks. In *Intl. Conf. on Cluster Computing*. IEEE Computer Society, 2003.

[17] Steven C. Pieper. Quantum Monte Carlo calculations of light nuclei. *Nuclear Physics A*, 751:516–532, 2005. Proceedings of the 22nd International Nuclear Physics Conference (Part 1).

[18] Lorna Smith and Mark Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9(2,3):83–98, 2001.

[19] Guy L. Steele Jr. Parallel programming and parallel abstractions in fortress. In *14th Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, page 157, 2005.

[20] Vinod Tipparaju, William Gropp, Hubert Ritzdorf, Rajeev Thakur, and Jesper L. Träff. Investigating high performance RMA interfaces for the MPI-3 standard. In *Proc. 38th Intl. Conf. on Parallel Processing (ICPP)*, September 2009.

[21] UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Laboratory, 2005.