MADNESS

```
#define
WORLD_INSTANTIATE_STATIC_TEMPLATES
#include
<world/world.h>


using
namespace
std;
using
namespace
madness;


class
Array
:
public
WorldObject<Array>
{
vector<double>
v;
public:
///
Make
block
distributed
array
with
size
elements
Array(World&
world,
size_t
size)
:
WorldObject<Array>(world),
v((size-1)/world.size()+1)
{
```

```
#define
WORLD_INSTANTIATE_STATIC_TEMPLATES
#include
<world/world.h>
using
namespace
madness;


class
Foo
:
public
WorldObject<Foo>
{
const
int
bar;
public:
Foo(World&
world,
int
bar)
:
WorldObject<Foo>(world),
bar(bar)
{
process_pending();
};


int
get()
const
{return
bar;};
};
```

Parallel Runtime

Last modification: 12/14/09

For more information please contact:
Robert J. Harrison
Oak Ridge National Laboratory
One Bethel Valley Road
P.O. Box 2008, MS-6367
Oak Ridge, TN 37831

email: harrisonrj@ornl.gov
tel: 865-241-3937
fax: 865-572-0680

Table of Contents

This documents provides an introduction to programming with the MADNESS parallel runtime and describes some of the implementation details. The runtime is used for the actual implementation of the MADNESS numerical capabilities and at least some understanding of the runtime is required by applications using the numerical tools. Also, the runtime may be used independently of the rest of MADNESS and will be separately distributed at some point in the future.

1 Overview

The MADNESS parallel programming environment combines several successful elements from other models and aims to provide a rich and scalable framework for massively parallel computing while seamlessly integrating with legacy applications and libraries. In particular, it is completely compatible with existing MPI and Global Array applications. All code is standard C++ tested for portability with a variety of compilers including the IBM, GNU, Intel, Portland Group, and Pathscale compilers. It includes

- Distributed sparse containers with one-sided access to items, transparent remote method invocation, an owner-computes task model, and optional user control over placement/distribution.
- Distributed objects that can be globally addressed.
- Futures (results of unevaluated expressions) for composition of latency tolerant algorithms and expression of dependencies between tasks.
- Globally accessible task queues in each process which can be used individually or collectively to provide a single global task queue.
- Serialization framework to facilitate transparent interprocess communication.
- Work stealing for dynamic load balancing (coming v. soon).
- Facile management of computations on processor sub-groups.
- Integration with MPI
- Active messages to items in a container, distributed objects, and processes
- Kernel-space threading for use of multi-core processors.

2 Motivations and attributions

There were several motivations for developing this environment.

- The rapid evolution of machines from hundreds (pre-2000), to millions (post-2008) of processors demonstrates the need to abandon process-centric models of computation and move to paradigms that virtualize, generalize or even eliminate the concept of process.
- The success of applications using the Charm++ environment to scale rapidly to 30+K processes and the enormous effort required to scale most process-centric applications.
- The arrival of multi-core processes and the consequent requirement to express much more concurrency and to adopt techniques for latency hiding motivate the use of light weight work queues to capture much more concurrency and the use of futures for latency hiding.
- The complexity of composing irregular applications in partitioned, global-address space (PGAS) models using only MPI and/or one-sided memory access (GA, UPC, SHMEM, co-Array) motivates the use of an object-centric active-message or remote method invocation (RMI) model so that computation may be moved to the data with the same ease as which data can be moved. This greatly simplifies the task of maintaining and using distributed data structures.

- Interoperability with existing programming models to leverage existing functionality and to provide an evolutionary path forward.
- The main early influences for this work were
  - Cilk (Kuszmaul, http://supertech.csail.mit.edu/cilk),
  - Charm++ (Kale, http://charm.cs.uiuc.edu),
  - ACE (Schmidt, http://www.cs.wustl.edu/~schmidt/ACE.html),
  - STAPL (Rauchwerger and Amato, http://parasol.tamu.edu/groups/rwergergroup/research/stapl), and
  - the HPCS language projects and the very talented teams and individuals developing these
    - X10, http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html
    - Chapel, http://chapel.cs.washington.edu
    - Fortress, http://fortress.sunsource.net

3 Programming environment and capabilities

The entire parallel environment is encapsulated in an instance of the class `World` that is instantiated by wrapping an MPI communicator. Multiple worlds may exist, overlap, and can be dynamically created and destroyed. Each world has a unique identity and the creation and destruction of a world are a collective operations involving all processes that participate in the world. To ensure full compatibility with existing MPI programs, the concept of place/location/process/rank is defined to be the same as MPI process.

The `World` class is intended to provide one-stop shopping for the parallel programming environment and, in particular, a uniform and consistent state that is always available. A local pointer to a world object may be passed to another process that is a member of the same world. During (de-)serialization the pointer is automatically translated so that in the remote process it correctly points to the local copy of the object. The class has members

- `mpi` - an instance of `WorldMPIInterface` that wraps standard MPI functionality to enable instrumentation, though you can obtain the wrapped MPI communicator and call the standard MPI interface directly.
- `am` - an instance of `WorldAMInterface` that provides low-level active message services. It is not intended for direct application use but rather as the framework upon which new parallel programming tools can be implemented.
- `taskq` - an instance of `WorldTaskQueue` that provides a light-weight task queue within each process that is accessible from all other processes in the world.
- `gop` – an instance of WorldGopInterface that provides additional global operations including collective operations that are compatible with simultaneous processing of active messages and tasks.

Within a world, distributed objects and containers (currently associative arrays or hash tables, with dense arrays planned) may be constructed.

3.1 Distributed or world objects

A distributed object has a distinct instance in each process of a world, but all share a common unique identifier. Thus, just like for a pointer to the World instance, a local pointer or reference to a distributed object is automatically translated during (de)serialization when sent to another process. Messages (method invocation) and tasks may be sent directly to such objects from any other process in the world.

Figure **??** contains the complete source for a program that creates two instances (`a` and `b` on line 20) of type `Foo` that serve distinct values from each MPI process. These are global objects but no synchronization is required before they may be used since messages to objects that are not yet locally constructed are automatically buffered. Lines 23 and 24 illustrate how messages (method invocation) can be sent to corresponding instances in any other process and how results are returned via a future that will hold the result when it finally becomes available. The

sample program immediately attempts to read the values (lines 26 and 27) and will wait until the value becomes available. Incoming active messages and any queued tasks are processed while waiting. The fence at line 29 ensures all data motion is globally complete before terminating the program. The comment on line 25 indicates the opportunity to do other work before forcing a future; indeed, this is their *raison d'être*. They facilitate asynchronous communication/computation and the hiding of both hardware and algorithmic latency. Futures may be probed without blocking to determine status and below we will see how to register callbacks in a future to be invoked when it is assigned.

Examining the implementation of `Foo` in figure **??**, it inherits from `WorldObject<Foo>` using the curiously recurring template pattern. This is because the base class needs the name of the derived class to forward remote method invocations. The `Foo` constructor initializes the base class and after finishing its own initialization (minimal here) invokes `process_pending()` to consume any messages that arrived before it was constructed. It is not possible to invoke this from the base class constructor since the derived class would not yet be fully constructed. Note that the `get()` method does not need to be modified from the natural sequential version – the send() template inherited from `WorldObject<Foo>` takes care of wrapping the return value in a future. Appropriate reference counting is used behind the scenes to ensure that locally allocated memory persists until the remote operation completes (i.e., the result of a remote operation may be safely discarded). Finally, the first line of the program requests that code for static members of `WorldObject<Foo>` be instantiated in the corresponding object file. This is a consequence of staying within standard C++ and not invoking any preprocessor to automate this process. In more sophisticated use, the ownership of a dynamically allocated `WorldObject` can be passed to the world (using a Boost-like shared pointer) for deferred destruction. At each global synchronization the world examines the reference count of registered objects to determine if any can be freed. Thus, actual destruction of such an object is deferred until the next global synchronization. In turn, this enables multiple such objects to be safely created and destroyed without any otherwise unnecessary global synchronization. No such sophistication is necessary in this example since we are happy with the introduction of the single global fence.

Figure **??** provides the complete implementation and example use of a crude, block-distributed array using the `WorldObject` functionality. First looking at the main program, on line 40 two distinct arrays are instantiated. Inside a parallel loop, the elements of the arrays are initialized without using locality at lines 44 and 45. The fence at line 47 ensures all data motion is complete before attempting to read from the array. This is only necessary with multiple readers or writers since a single process is guaranteed a sequentially consistent view due to the world active-message layer guaranteeing in-order processing of messages. Reading an array element (lines 52 and 53) returns a `Future<double>` that will hold the result when it finally becomes available.

Turning to the implementation of the `Array` class in Figure **??**, reading and writing elements is immediate if the element is local (lines 22 and 29), otherwise `WorldObject<Array>.send()` is used to forward the request to the `Array` object in the owning process, passing arguments as necessary. Attentive readers will have noticed that the `write()` method returns `Void` rather than `void`. This is merely to simplify the current implementation that would otherwise require specialization of most templates to handle `void` results. Once the interface has stabilized this design choice will be reconsidered. Futures of type `void` and `Void` are minimal stubs and cause no communication.

There are two main restrictions on methods that are invoked remotely. First, the arguments must be values or constant references and must be serializable (see below). Pointers to `World`, or pointers or references to `WorldObjects` are automatically translated to refer to the appropriate remote object, but any other pointers are the responsibility of the application (though their translation via serialization may also be automated). Second, the method itself must not block, e.g., by forcing another future. This restriction can be greatly relaxed, but is presently enforced to avoid potential stack overflow and other problems due to deeply-nested invocation of handlers.

- - - - - - - - - - - STOPPED HERE ... lots more to do .....

3.2 Distributed or world containers

Distributed containers are distributed objects specialized to provide the services expected of a container and to pass messages directly to objects in the container. The latter enables non-process centric parallel computation in the sense that all messaging is between objects addressed with user-defined names and with transparent association of names to processes. BLAH BLAH ...

The only currently provided containers are associative arrays or maps. The underlying sequential container on each process is either the GNU `hash_map` or the TR1 `unordered_map`. A map generalizes the concept of an array

(that maps an integer index in a dense range to a value) by mapping an arbitrary key to a value. This is a very natural, general and efficient mechanism for storing sparse data structures. The distribution between processes of items in the container is based upon a function which maps the key to a process. The default mapping is a pseudo-random uniform mapping based upon a strong hash function, but the user can provide their own (possibly data-dependent) operator to control the distribution.

Although it presently the case that all processes agree on the mapping of a key to a process, this does not have to be the case. The implementation is designed to support forwarding of remote requests though this code is not yet enabled or tested. The point is that it may be effective to perform local redistributions of data in order to address load or memory problems rather than globally changing the map which must be deferred until a synchronization point.

The keys and values associated with containers must be serializble by the MADNESS archive mechanism.

Please refer to world/archive/archive.h and documentation therein for information about this. In addition, the keys must support

- testing for equality, either by overloading \c == or by specializing \c std::equal_to<key_type>, and
- computing a hash value by invoking \c madness::hash(key), which can be done either by providing a member function with signature

\code
  hashT hash() const;
\endcode

or by specializing \c madness::Hash<key_type>.

hashT is presently an unsigned 32-bit integer. MADNESS provides hash operations for all fundamental types, and variable and fixed dimension arrays of the same. Since having a good hash is important, we are using Bob Jenkin's "lookup v3" hash[1].

Here is an example of a key that might be used in an octtree.

```
struct Key {
    typedef unsigned long ulong;
    ulong n, i, j, k;
    hashT hashval;

    Key() {};

    // Precompute the hash function for speed
    Key(ulong n, ulong i, ulong j, ulong k)
        :  n(n), i(i), j(j), k(k), hashval(madness::hash(&this->n,4,0)) {};

    hashT hash() const {
        return hashval;
    };

    template <typename Archive>
    void serialize(const Archive& ar) {
        ar & n & i & j & k & hashval;
    }

    bool operator==(const Key& b) const {
        // Different keys will probably have a different hash
        return hashval==b.hashval && n==b.n && i==b.i && j==b.j && k==b.k;
    };
};
```

To be added

- discussion of chaining hashes using initval optional argument

[1] http://www.burtleburtle.net/bob/c/lookup3.c

- discussion of overriding the distribution across processes

3.2.1 Tasks, task queues, futures, and dependencies

This is the heart of the matter ...

3.2.2 Serialization

Serialization ...BLAH BLAH ...

4 Recommended programming paradigms

BLAH BLAH ..

The recommended approaches to develop scalable and latency tolerant parallel algorithms are either object- or task-centric decompositions rather than the process-centric approach usually forced upon MPI applications. The object-centric approach uses distributed containers (or distributed objects) to store application data. Computation is expressed by sending tasks or messages to objects, using the task queue to automatically manage dependencies expressed via futures. Placement of data and scheduling or placement of computation can be delegated to the container and task queue, unless there are specific performance concerns in which case the application can have full knowledge and control of these.

Items in a container may be accessed largely as if in a standard STL container, but instead of returning an iterator, accessors instead return a Future<iterator>. A future is a container for the result of a possibly unevaluated expression. In the case of an accessor, if the requested item is local then the result is immediately available. However, if the item is remote, it may take some time before the data is made available locally. You could immediately try to use the future, which would work but with the downside of internally waiting for all of the communication to occur. Much better is to keep on working and only use the future when it is ready.

Aside:

- To avoid a potentially unbounded nested invocation of tasks which could overflow the stack and also be the source of live/deadlocks, new tasks are not presently started while blocking for communication. This will be relaxed in the near future which will reduce the negative impact of blocking for an unready future as long as there is work to perform in the task queue.
- Once fibers or user-space threads are integrated, multiple tasks will always be scheduled and blocking will merely schedule the next fiber.

By far the best way to compute with futures is to pass them as arguments to a new task. Once the futures are ready, the task will be automatically scheduled for execution. Tasks that produce a result also return it as a future, so this same mechanism may be used to express dependencies between tasks.

Thus, a very natural expression of a parallel algorithm is as a sequence of dependent tasks. For example, in MADNESS many of the algorithms working on distributed, multidimension trees start with just a single task working on the root of the tree, with all other processes waiting for something to do. That one task starts recursively (depth or breadth first) traversing the tree and generating new tasks for each node. These in turn generate more tasks on their sub-trees.

The execution model is sequentially consistent. That is, from the perspective of a single thread of execution, operations on the same local/remote object behave as if executed sequentially in the same order as programmed. This means that performing a read after a write/modify returns the modified value, as expected. Such behavior applies only to the view of a single thread – the execution of multiple threads and active messages from different threads may be interleaved arbitrarily.

4.1 Abstraction overhead

Creating, executing, and reaping a local, null task with no arguments or results presently takes about 350ns (Centos 4, 3GHz Core2, Pathscale 3.0 compiler, -Ofast). The time is dominated by `new` and and `delete` of the task structure, and as such is unlikely to get any faster except by caching and reusing the task structures. Creating and then executing a chain of dependent tasks with the result of one task fed as the argument of the next task (i.e., the input argument is an unevaluated future that will be assigned by the next task) requires about 1us per task (3 GHz Core2).

Creating a remote task adds the overhead of interprocess communication which is on the scale of 1-3us (Cray XT). Note that this is not the actual wall-time latency since everything is presently performed using asynchronous messaging and polling via MPI. The wall-time latency, which is largely irrelevant to the application if it has expressed enough parallelism, is mostly determined by the polling interval which is dynamically adjusted depending upon the amount of local work available to reduce the overhead from polling. We can improve the runtime software through better agregation of messages and use of deeper message queues to reduce the overhead of remote task creation to essentially that of a local task.

Thus, circa 1us defines the granularity above which it is worth considering encapsulating work (c.f., Hockney's n1/2). However, this is just considering the balance between overhead incurred v.s. useful work performed. The automatic scheduling of tasks dependent upon future arguments confers additional benefits, including

- hiding the wall-time latency of remote data access,
- removing from the programmer the burden of correct scheduling of dependent tasks,
- expressing all parallelism at all scales of the algorithm for facile scaling to heavily multi-core architectures and massively parallel computers, and
- virtualizing the system resources for maximum future portability and scalability.

Available memory limits the number of tasks that can be generated before any are consumed. In addition to application specific data, each task consumes circa 64 bytes on a 64-bit computer. Thus, a few hundred thousand outstanding tasks per processor are eminently feasible even on the IBM BG/L. Rather than making the application entirely responsible for throttling it's own task production (which it can), if the system exceeds more than a user-settable number of outstanding tasks, it starts to run ready tasks before accepting new tasks. The success of this strategy presupposes that there are ready tasks and that these tasks on average produce less than one new task with unsatisfied dependencies per task run. Ultimately, similar to the Cilk sequentially-consistent execution model, safe algorithms (in the same sense as safe MPI programs) must express tasks so that dependencies can be satisfied without unreasonable expectation of buffering.

In a multiscale approach to parallelism, coarse gain tasks are first enqueued, and these generate finer-grain tasks, which in turn generate finer and finer grain work. [Expand this discussion and include examples along with work stealing discussion]

Discussion points to add

- Why arguments to tasks and AM via DC or taskQ are passed by value or by const-ref (for remote operations this should be clear; for local operations it is to enable tasks to be stealable). Is there a way to circumvent it? Pointers.
- Virtualization of other resources
- Task stealing
- Controlling distribution in containers
- Caching in containers
- Computing with continuations (user space fibers)
- Priority of hints for tasks

5 C++ Gotchas

5.1 Futures and STL vectors

A common misconception is that STL containers initialize their contents by invoking the default constructor of each item in the container since we are told that the items must be default constructible. But this is *incorrect*. The items are initialized by invoking the copy constructor for each element on a *single* object made with the default constructor. For futures this is a very bad problem. For instance,

```
vector< Future<double> > v(3);
```

is equivalent to the following with an array of three elements

```
Future<double> junk;
Future<double> v[3] = {junk,junk,junk};
```

Since the Future copy constructor is by necessity shallow, each element of `v` ends up referring to the future implementation that underlies `junk`. When you assign to an element of `v`, you'll also be assigning to `junk`. But since futures are single assignment variables, you can only do that once. Hence, when you assign a second element of `v` you'll get a runtime exception.

The fix (other than using arrays) is to initialize STL vectors and other containers from the special element returned by `Future<T>::default_initializer()` that if passed into the copy constructor will cause it to behave just like the default constructor. Thus, the following code is what you actually need to use an STL vector of futures

```
vector< Future<double> > v(3,Future<double>::default_initializer());
```

which, put politely, is ugly. Thus, we provide the factory function

```
template <typename T>
vector< Future<T> > future_vector_factory(std::size_t n);
```

that enables one to write

vector< Future<double> > v = future_vector_factory<double>(3);.

6 Multi-threading, the task queue, active messages and SMP parallelism

This design is in flux but the overall objectives are to provide a model and set of abstractions that are compact and well defined so that they may readily understood and reasoned about, yet rich enough to achieve compact and high-performance expression of most algorithms.

Key design points:

- MADNESS can be configured to build either with or without threads. If configured with threads the number of active threads can be adjusted at runtime.
- The only guaranteed source of SMP/multi-threaded concurrency is from the task queue.
- Active messages from a given source are executed sequentially in the order sent – this is so that they may be used to maintain data structures with no additional logic necessary for the application to enforce sequential consistency.
- Active messages from distinct sources may be executed in any order or concurrently. Presently, one thread is devoted to handling all active messages so there is no concurrency at all within the active message queue, but this will change once the active message queue and task queues are unified.
- The main thread of execution may execute concurrently with the active message and task pool threads. Presently, the main thread also acts as the active message thread and in the absence of other work will execute tasks from the queue.
- All World interfaces are thread safe (are there any exceptions?). In particular, WorldContainers and the provided functionality of WorldObjects are thread safe as is the reference counting in SharedPtr.

- Parallel applications written using only MADNESS World constructs (tasks, containers, futures, and active messages) will not need additional mechanisms for SMP concurrency or synchronization. However, some applications will have additional shared data structures and provided are classes for facile use of threads, mutexes and locking pointers. At the lowest level are a portable (limited) set of atomic operations on integer data that are presently used to compose the SharedPtr and Mutex classes.

How to think about all this?

Where possible compose your application in terms of tasks with dependencies via futures. This provides the maximum concurrency and as additional task attributes are introduced will enable the most efficient scheduling of work. The overhead of making, executing and reaping a task is about 1us, so a task's runtime should be bigger than circa 10us unless there is additional latency (algorithmic or communication) that will be hidden by making a task. If tasks are too long there may be load-balancing problems. Unless it is unduly cumbersome to split the problem into small tasks there is no reason to make tasks larger than circa 1ms (exception to this is pushing/stealing tasks that may have communication overhead).

When to use active messages rather than a task? The main benefits of AM are sequential consistency (from the perspective of the sending process) and high priority for their execution. To keep the real AM latency to a minimum (which means you need less concurrency to hide the latency) AM handlers should be lightweight. Presently, MADNESS does no aggregation of messages but this is something that is clearly needed and would, for instance, reduce the overhead of creating many remote tasks to about that of local task creation.

7 Acknowledgments

DOE NSF DARPA ORNL NCCS UT

8 References