Parallel Programming Models for MADNESS

Vlad Slavici

PROBLEM
Madness needs a parallel programming model that can be implemented efficiently on various parallel architectures. The programming model should not change between architectures, only the implementation of the model can change. This means that a piece of code can be run with no modification on various architectures (mostly of interest are *massive computer clusters* and *GPGPUs*).

IMPORTANT: The central issue with the current Madness model, called the "task" model, is the lack of separation between data access and computation. The current model is based on executing many independent tasks in parallel, but, since these tasks combine data access and computation, it is difficult to optimize data access in particular and to port algorithms to various architecture (dataflow and control flow organization becomes difficult). Since the current "task" model is not easily compatible with CPU-GPU clusters, a separate "task" model would need to be designed for CPU-GPU clusters. We propose an alternative in this document: a higher-level Continuation Passing Style (CPS) model which can be used with no top-level modification on any architecture. This CPS model can have different low-level implementations for each parallel architecture.
Being able to organize dataflow and control flow is necessary for a portable parallel programming model. The user API should be high-level, detached from any particular hardware architecture and flexible enough so that dataflow and control flow can be easily controlled at some level. Modifying the rules for dataflow and control flow is the key to obtaining good performance across various parallel architectures.
One requirement for the general programming model is that it can emulate the current "task" model relatively closely, in control flow as well as performance.

EXAMPLES OF KERNELS
It is important to see a couple of examples of widely used kernels in Madness in order to be aware of the data access patterns used and the important bottlenecks in the algorithms.
Currently, most kernels implemented on top of Madness are CPU-limited. However, it is desirable that a general programming model try to address, as much as possible without losing generality, many potential bottlenecks for kernels that will be written in the future.
Reconstruct. The inputs of the "reconstruct" kernel are: a tree of coefficients, `coeffs`. Each node in the tree is a `(key, coefficient)` pair, and an initial `(key, tensor)` pair.
The algorithm is a recursive one:

```
void reconstruct(key, tensor){
  1. find entry = (key, some coefficients) in coeffs;
  2. process entry and update entry in coeffs;
  3. for each child node of entry in coeffs, compute key' and tensor'
  4.     call reconstruct(key', tensor')
}
```

We also know, as an important detail, that operations at a certain level in the tree of coefficients generate recursive calls only to the immediate next level in the tree.
In the default Madness implementation, steps 2 and 3 are the bottleneck, and they are CPU-limited.

"reconstruct" operates on a compressed tree (a tree in which leaves do not have coefficients – all coefficients are stored in non-leaf nodes) and pushes the coefficients from the inner nodes to the leaves (of course, the coefficients change during the operation).
Compress. The inputs of the "compress" kernel are: a tree of coefficients, `coeffs`. Each node in `coeffs` is a `(key, coefficient)` pair, an initial key, and two initial boolean flags.
The algorithm is a recursive one, more complex than the reconstruct recursion:

```
tensor compress(key, flag1, flag2){
  1. find entry = (key, some coefficients) in coeffs;
  2. for each child (key', something) of entry{
  3.     v[key'] = compress(key', flag1, flag2);
     }
  4. based on (all v[key'] from 3)
  5.       update entry in coeffs;
  6.       compute tensor-value;
  7. return tensor-value;
}
```

Processing is a CPU bottleneck here as well. Also, as opposed to "reconstruct", which is a tail recursion, "compress" is an n-ary recursion which requires two passes over the implicit graph of the problem: the first one is top-down (step 3), the second one is bottom-up (steps 4 – 7).

Considerations about Typical Madness Kernels
A tree, `coeffs`, is contained in a *container*, which is a distributed data structure (a distributed hash table). The MPI process that owns a container entry is given by a Process Map, which is shared among all processes. The location of an entry in a process' address space is given by applying a hash function to the entry's key. Given any node (not necessarily the root node of a subtree), one can use the Process Map to derive, from the node's key, the MPI process rank where that node is stored.
The container can maintain some locality by storing a subtree in the memory of a single MPI process (by using an adequate Process Map).
Conceptually, the "reconstruct" algorithm simply requires a top-down exploration of the tree in any order. The container locality (subtree locality) enables further efficiency.
Hence, in "reconstruct", an MPI process will explore a subtree belonging to it. When the MPI process detects that a child belongs to a different process, the process will send a message to the remote process to continue to update the child node and its descendants. The original process then goes on to process the remaining nodes in its subtree.
Note that the tree is unbalanced, and so the computations on some subtrees will be completed before the computations on other subtrees. Hence, it is important for efficiency reasons to treat this as a dataflow problem, and to avoid synchronization points that wait for the slowest member.
The MPI processes are distributed across the computer hosts, usually with one MPI process per computer host. An MPI process uses a pool thread for task processing, so that no host is starved for work.

"compress" operates on a reconstructed tree, by pushing coefficients from the leaves up to the inner nodes.

IMPORTANT: Note that for the "reconstruct" algorithm an easy task-based parallel implementation can be produced: keys can be assigned to execution treads independently of one another and independent threads can work on independent parts of the implicit call graph. Only one synchronization call is necessary, at the end (when there are no more tasks to be generated). The same is true for the top-down scan of the call graph in "compress". But the bottom-up scan of compress needs a synchronization call at each level of the call graph, because the update pattern is update from many to few (lines 4 – 7), so a task cannot return until it received updates from all child tasks. The synchronization here does not need to be per-container. Tasks can mark themselves ready or waiting, but this complicates the model and does not work in the cases

in which a parent task does not know all of its child tasks (but maybe this does not happen in Madness). Either way, the number of synchronization calls needed is probably small enough for the desired efficiency in the CPS model, which is discussed below.

POSSIBLE APPROACHES

1. **The functional "MapReduce-like" programming model**, described in the document "A MapReduce-like Model for Madness". There are a number of issues with this model, the most important of which seems to be the need for too many synchronization calls, even if they are per-container. Other issues are the need to produce a new container at each call to a MapReduce-like operation, which requires a lot of data to be duplicated and the need to scan some data structures multiple times, because of the insurmountable lack of primitives (e.g. map1Update2 would be needed, which gets too complicated for the user). This model corresponds to a BFS exploration of the call graph.

2. **The Continuation Passing Style (CPS) model**, a la Roomy, but with optional synchronization after each call to a CPS function: in Madness many kernels only require synchronization at a few points in the computation, not after each call. This model corresponds to a batch-parallel DFS traversal of the call graph.

   IMPORTANT: The MapReduce-like model suffers from a number of intrinsic performance issues. The CPS model addresses these issues by trying to achieve:

   (a) Minimization of the number of synchronization calls (ideally needing as few sync calls as in the "task" model)

   (b) Modifying data structures in place.

   (c) Minimize data access redundancy (access a piece of data as few times as possible).

   The CPS model has other goals as well:

   (a) Separation of data access from computation, for increased flexibility of the underlying implementation

   (b) Easy to use - the top level api is not a very abrupt departure from the "task" model

   (c) portability across many parallel architectures with no performance penalty

   Continuation Passing Style was introduced by Sussman and Steele in 1975 and it refers to a programming style in which functions do not return to the caller, but rather pass control explicitly to a continuation, which is a function passed as an argument to the original function (see the Wikipedia article: http://en.wikipedia.org/wiki/Continuation-passing_style).

   In this document, the meaning of Continuation Passing Style is very related to the meaning in functional programming, but with a few distinctions: a function can pass logical control threads to multiple continuations and still continue to execute code itself – this model uses continuations as objects (see the above cited Wikipedia article). Also, CPS was used as the main idea behind many past parallel programming models.

   Pipelining in the Sense of Databases: implementations for Parallel architectures often benefit from SIMD parallelism. To take advantage of SIMD parallelism, the same function must be applied to multiple data elements in parallel. The "MapReduce-like" model exhibits SIMD parallelism, but at the cost of decreased efficiency. Roomy makes use of SIMD parallelism, but it needs synchronization after each massively-parallel batch operation. This is ok for parallel disk-based computing and the typical space-limited applications Roomy was designed for, but is inefficient for typical Madness computations. In typical Madness algorithms and kernels it would be too expensive to synchronize after each BFS level in the callgraph. So we need a solution that provides enough batch parallelism for SIMD operations

while at the same time, minimizes the number of synchronizations. This is provided by updating the Roomy model to use fewer synchronizations. By acknowledging that typical Madness applications rarely need synchronization, we modify the Roomy model by using pipelining in the sense of databases. While in Roomy users would synchronize for each BFS level of tasks, in the CPS model they would synchronize roughly once or twice per algorithm. This is possible because typical Madness applications do not need duplicate detection, so delaying synchronization will not lead to an increase in memory usage, which makes it possible to work on some tasks at a certain level (even though not all tasks at that level are available) and generate tasks at the next level. Once there are enough tasks at the next level for SIMD parallelism to be efficient, these tasks can be processed without waiting for all the same-level tasks to become available. So batches of tasks at multiple levels can be processed at the same time, which is different from Roomy. To program in this model, users will define their own continuations, but the API will only allow passing control via two operations: `access` and `update`. Invoking any of these two functions requires specifying a location in a WorldContainer (distributed hash table) and a closure, both being the user's responsibility. This will be implemented by having the two API functions be wrappers around executing the function in the passed closure on the data in the closure and the data at the specified location in the specified WorldContainer, which can also include more processing. Before a closure is executed, the parameters it needs have to be ready. While waiting for the parameters, the closure is in "waiting" state. Once the parameters are ready, the closure is marked as "ready" and can be executed at any time, anywhere. For many parallel architectures, data needs to be processed in batches. Once we have a batch of ready closures, they can be executed. Batches of ready closures at many breadth-first levels in the callgraph can be executed at the same time, as long as no synchronization is necessary after a callgraph level, and this is usually the case in Madness applications. This is similar to Roomy in the sense that when a batch of ready closures is full, it is being send to the destination host immediately. It is different from Roomy in the fact that, in Roomy, all the batches at the same level have to arrive to the host before the host can execute them; in Madness once a batch arrives it can be executed, because it is usually independent from all other batches. The user must explicitly synchronize the computation when a level has to be fully ready before another level of tasks can start execution, but this is rare. When synchronization is not needed, pipelining as in databases is provided.

A WorldContainer will have a few methods: map, access, update, fence. Functors (closures) will be passed to each access/update call. These functors need to be registered with the WorldContainer in order to be used.

A call to update will look something like:

```
dc.update(Functor, closure);
```

These requests will be buffered in per-container-per-functor buffers. Requests will be executed when the buffer is full or when there are no more outstanding requests to be received for that buffer.

"Reconstruct" can be implemented in this model with only one fence per multiple containers. "Compress" requires one fence per multiple containers for the top-down scan and one fence per multiple containers per level in the bottom-up scan. One fence per multiple containers is much more lightweight than a global fence.

This model greatly alleviates the need for too many fences, while still keeping as much as possible of the MapReduce-like model. The improvement in performance comes from two differences when compared to the MapReduce-like model:

- a large part of many Madness kernels does not require that a full level of a tree be consistent to operate on a part of that tree-level that is consistent; and large parts of tree operations can be split into operations on subtrees, thus exhibiting task parallelism. What is meant here by task parallelism is that a thread of execution can work independently on sub-problems without communicating, or communicating at a minimal level, with other execution threads.

- modifications of WorldContainers are done in place (side effects) This means that no data is accessed more times than minimally required by the kernel operation.

Example: **Reconstruct**. Map over `tasks[0]`. In the map body call `coeffs.update`, and send information along. In the body of the update extract information from the specific node in the `coeff` tree, update the node, do some processing, and call `tasks[1].update` to generate new tasks. These update continuations will insert data in tasks[1] and then reiterate through the control flow cycle:

$$\texttt{tasks[0]} \xrightarrow{map(coeffs.update(functor_0))} \texttt{coeffs} \xrightarrow{tasks[1].update(functor_1)} \texttt{tasks[1]} \xrightarrow{coeffs.update(functor_0)} \texttt{coeffs} \xrightarrow{tasks[2].update(functor_1)} \texttt{tasks[2]} \ \ldots$$

Parallel CPS (Continuation Passing Style) is a series of cascading updates initiated by a `map` operation, and kept flowing by updates/accesses. In the case of "RECONSTRUCT", these cascading operations need no synchronization, except at the end of the cascade, when ensuring the consistency of all data structures updated in the algorithm is necessary.

This model enables massive parallelism (so we don't lose the ability of MapReduce) for vectorization, while relaxing the condition that the graph traversals are BFS. Batch processing of tasks is still there, but, if not required by the algorithm, a batch operation doesn't have to synchronize and wait until all operations are performed on a WorldContainer; rather, a batch operation can be split into sub-batches, each acting independently on a part of the container, and generating updates to parts of another WorldContainer.

There are cases in which a per-container synchronization is necessary, to ensure consistency, but they are much less frequent than in MapReduce.

The CPS model can also control bounded memory buffer requirements, since batches can be split into arbitrarily small sub-batches. Elements in each sub-batch will be processed in parallel.

Here is the CPS model pseudo-code for the "reconstruct" operation:

```
  void reconstruct(key, tensor){
    coeffs.update(key, &update_func, tensor); //very similar to invoking a task,
//but this time the data is specified (key) and not the MPI rank (owner(key))
    coeffs.fence();
  }

  void update_func(key, node, tensor){
    process key, node, tensor;
    generate array child_tensor; //all computation
    for each child of key{
      coeffs.update(child, &update_func, child_tensor[child]);
    }
  }
```

Note that the update function does no data access, so vectorization for GPUs is achieved by executing multiple update_func simultaneously.

Example: **Compress**.

(a) *Top-down* Map over `tasks[0]`. In the map body call `coeffs.update`, and send information along. In the body of the update process data, update node and call `tasks[1].update` for new task generation and `tensors[0].update` to create the tensor tree top-down. The updates that generate new tasks continue the cycle. After reaching the bottom level of the tree, fence over the last tensors container and over coeffs.

$\texttt{tasks[0]} \xrightarrow{map(coeffs.update(functor_0))} \texttt{coeffs}$
$\texttt{coeffs} \xrightarrow{tasks[1].update(functor_1)} \texttt{tasks[1]}$
$\texttt{coeffs} \xrightarrow{tensors[0].update(functor_2)} \texttt{tensors[0]}$
$\texttt{tasks[2]} \xrightarrow{map(coeffs.update(functor_0))} \texttt{coeffs}$
...
fence last tasks, coeffs, last tensors

Above is an inlining of the DAG of the computation.

(b) *Bottom-up*

`current level = last level`
while `current level > 0`{

$\texttt{tensors[current level]} \xrightarrow{map(coeffs.update(functor_3))} \texttt{coeffs} \xrightarrow{tensors[currentlevel-1].update(functor_4)} \texttt{tensors[current level - 1]};$

fence `tensors[current level - 1]`

current level = current level - 1

}

The bottom-up scan needs a fence at each level of tensors, because multiple updates are being sent to the same node, and they have to be aggregated before continuing.
Here is the pseudocode for the "compress" kernel:

```
void compress(key, flag1, flag2){
  coeffs.access(key, &top_down, flag1, flag2, 0, 0);
  coeffs.fence();
  level = last_level;
  while (level > 0){
    tensors[level].map(&send_parent);
    tensors[level-1].fence();
  }
  tensors[0].map(send_coeffs);
  coeffs.fence();
}

void top_down(key, node, flag1, flag2, level, parent){
  tensors[0].update(key, &create_tensornode, all children, flag1, flag2, parent);
  for each child of key{
    coeffs.access(child, &top_down, flag1, flag2, level+1, key);
  }
}

void create_tensornode(key, children, flag1, flag2, parent){
  tensors[0].insert(key, (children, flag1, flag2, parent));
}

void send_parent(key,(children, flag1, flag2, parent)){
```

```
    tensor = local_processing(key,(children, flag1, flag2, parent));
    tensors[level-1].update(parent, &set_myparent, key, tensor); //just sets myparent
    coeffs.update(key, &set_node, local info);//just sets node
}
```

A mechanism is needed for per-container synchronization when there are multiple levels of outstanding messages.

Comparison to the Roomy Model

Two important questions must be answered:

(a) How can we use fewer syncs in Madness than in Roomy?

(b) Why do we need to worry about fewer syncs?

The answer to question (a) is: because of the intended architecture and the typical application. Madness is intended to run on the distributed RAM of massive clusters whereas Roomy is intended for parallel disks on data intensive computations. Mainly because the typical application in Roomy relies heavily on delayed duplicate detection it is often inefficient to work only on part of the data structure (because this can quickly lead to a state explosion due to duplicate work – some duplicates are not detected in time and this propagates to the next steps in the program). However, if applications did not perform delayed duplicate detection as much, then it would be conceivable to reduce the number of syncs. However, typical Madness applications do not need duplicate detection at all. They exhibit more task parallelism. So there is no need, in many cases, to wait for a container (distributed hash table) to be overall consistent, because working on a part of it can be done independently of work on other parts of it. Still, there are cases in which syncs are needed: an example is the bottom-up scan in "compress", in which multiple child nodes update a parent node. The parent must then wait for all updates from the child nodes to arrive. This is a simple case, in which synchronization can be eliminated if some extra logic is embedded in the update task (the update function checks whether the parent has received updates from all children). But in more general cases (which might not appear in Madness) synchronization is necessary.

The answer to question (b) is also the typical application and architecture. Typical Madness kernels are CPU limited (in the future there might be some data limited ones) and they can exhibit memory locality, as opposed to the problems Roomy is used for, where data access is the limiting factor. Having too many synchronizations can push network access to be a bottleneck, too. Limiting the exploration of the problem implicit graph to BFS delays the exploration of some graph areas which would be available immediately otherwise. This limitation is both a product of some nodes possibly being slow and of the time needed to frequently sync across many nodes. If syncs are removed, some nodes will be available sooner to other parts of the computation. Also, if processing of local data starts to become comparable to the time it takes to fully sync, the performance hit is unacceptable.

Implementation on a GPGPU connected to a CPU

On a GPGPU we need all the work to be done by the GPU and the data access by the CPU. This is because the entire data fits in RAM on the CPU, but not in the GPGPU memory. Control flow needs to switch between the two for the same logical thread and computation has to efficiently run on both in parallel.

The sub-batches are especially useful in this CPU-GPGPU setting. Data accesses have to be done entirely on the CPU because random access in RAM is fast, whereas the latency between the GPGPU and RAM is on the order of a microsecond.

Accesses and updates are buffered on the GPU memory and implemented in a few steps. These steps separate actual data access, which is done on the CPU, and processing, which is done on the GPU.

Access operations are implemented by the following sequence of steps:

- the GPU generates accesses (a functor and a location in a WorldContainer)
- the closure of the functor and location are saved to a buffer on the GPU
- when the buffer is full, the closures are sent to the CPU
- access the data from the WorldContainer locations specified in the closures on the CPU
- send the retrieved data and the functor batch by batch to the GPGPU
- the GPGPU runs the functors using the data and replies to the CPU that it is ready for the next batch in the same access operation.

Any generated accesses and updates are buffered on the GPU and, when buffers get full, they are being sent to the CPU.

Update operations are implemented by the following sequence of steps:

- the GPU buffers generated updates (a functor and a location in a WorldContainer)
- the buffered updates are sent to the CPU
- for each received update, data is accessed from the target WorldContainer and saved in a buffer, together with the update functor, as a closure
- update closures buffers are sent to the GPGPU
- the functors are called on the data on the GPGPU
- update functors return results, which are buffered and sent back to the CPU
- a notification that the current buffer was processed is also sent to the CPU
- updating the container on the CPU with the received results
- accesses and updates generated by the functors are buffered on the GPU a.s.o.

Programming for the CPU-GPU follows a client-server model, in which the GPU is the client and the CPU is the server. The CPU only initiates control flow and answers to requests from the GPU (collects necessary data from RAM). Almost all processing is done on the GPU, with closures (functors with data), while the CPU is only responsible for data accesses and sending data back to the GPU.

Implementation on a cluster of CPU-GPGPUs

In the one CPU-GPGPU model all data accesses went to the CPU and all processing went to the GPU. When the architecture is a cluster of CPU-GPGPUs, data accesses can also be sent over the network. By using the CPS model, replies to data requests from a GPGPU can end up being processed by another GPGPU, and most likely by multiple GPGPUs. We can now view the architecture as a massive monolithic CPU and a massive monolithic GPU that are connected by a very large bus.

Update operations generated by a GPU will be processed as follows:

- buffer update functors and data location requests
- send buffer to local CPU
- send buffered requests to their owner CPUs over the network
- CPU receives buffers from other CPUs
- CPU accesses requested data locations from the WorldContainer in local RAM

- closures with functors and accessed data are buffered
- buffer is sent to local GPGPU
- GPGPU processes closures and generates accesses/updates

There are three issues that need to be addressed for CPU-GPGPU programming, for architectures like Titan:

(a) As few synchronizations as possible – done by using the CPS model

(b) Controlling the working set of the computation – bounded memory – working on sub-batches

(c) Fully enable vectorization – exploiting SIMD parallelism on sub-batch elements

Implementation: A Protocol based on a Dataflow Model with Task-Stealing (from Gene)
An ideal dataflow model would have each MPI process (assuming one process per host) retain a work queue. It would also have a special thread that receives messages as they come in. When a node discovers a child node in a different process, it generates a WORK message. The corresponding work is added to the work queue of that process.

1. When the work queue rises above a certain threshold, the communication thread will also send out an ASSISTANCE request to a random remote process requesting assistance. It will continue to send out ASSISTANCE requests at regular intervals until its work queue is below the given threshold.

2. When the communication thread receives a request to share the work, and if its own work queue is empty, then it will send a SHARE message back requesting to share the work. If its own work queue is not empty, then it will not reply to the ASSISTANCE request.

3. Upon receiving a SHARE message, the communication thread will reply either with SHARE-ACK (and send half of its work queue), or else reply with SHARE-NACK (not enough work to steal; the work queue is below threshold).

Note that the random ASSISTANCE request does not take account of locality within the network topology. One could also imagine two thresholds. When the work queue rises above the first threshold, one sends an ASSISTANCE request to a remote host that is nearby in the topology. When the work queue is above the second threshold, one sends an ASSISTANCE request to a remote host that is far away in the topology.
Conceptually, the "compress" algorithm simply requires a bottom-up exploration of the tree. All child nodes must be processed before the parent node can be processed. This access pattern automatically takes advantage of container locality (subtree locality).
An important advantage of this dataflow model, is that it lends itself nicely a GPU. The data is likely to live within global GPU memory, and is equally accessible to all threads. In GPUs, the most expensive operations are access to CPU host memory, global synchronization, and access to global memory — roughly in that order. This dataflow model can be organized so that synchronization in global memory is mostly not needed. Messages are posted to a "mailbox" in global memory. But it's okay for one thread to overwrite an ASSISTANCE request with its own request. Only the most recent ASSISTANCE request is processed, but since these requests are sent randomly anyway, there is no violation of correctness of the algorithm. When an ASSISTANCE request is accepted, a separate address is provided for the SHARE-ACK or SHARE-NACK reply, so that there is no further collision.
For GPUs, the dataflow model must be extended to the case when there is a request to process some nodes of a container that reside only in the CPU memory and not in the GPU global memory. At regular intervals, that CPU memory must be brought into the GPU. There are several possible implementations. The simplest one is to have a single GPU kernel process all of the nodes in its GPU global memory, and then finish the kernel (equivalent to a synchronization). A more sophisticated implementation might allow the kernel to exchange global memory with CPU host memory as it needs to process additional nodes not formerly in the GPU global memory.

NOTE FROM GENE: In Madness, I would still be worried about load balancing. A dataflow has absolutely the minimum possible number of synchronization points. I believe the dataflow model described above for "reconstruct" would also work for "compression".
One could use futures in the obvious way, here. But to keep the implementation simple, I might suggest not using futures on the first iteration. Instead, one can stay with a pure dataflow model. A process does not begin to work on a subtree until all the children of that subtree have already been processed, and their results are known.
NOTE FROM GENE (for my own curiosity): I assume (based on what Vlad told me) that one could do "compress" purely bottom-up. If a node has no remote child nodes, then it must be a leaf node. If a node always knows where its parent is, then work could begin bottom-up, by working purely bottom up. Since an initial top-down phase will be the smaller part of the computation, maybe the overhead for top-down doesn't matter anyway.
FROM VLAD: Indeed, "compress" can be done purely bottom-up. One would just need to know what are all the leaves of the tree.