A MapReduce-like Programming Model for MADNESS

- using a breadth-first traversal of call graphs -

Vlad Slavici

1 Motivation

A significant part in the mra kernels in M-A-D-N-E-S-S is implemented in terms of operations on WorldContainers. A WorldContainer is a distributed hash table, made up of locally- stored hash tables on each compute node and a global process map, which determines which compute node owns a (key, value) pair.

In the current model (the "task" model), a kernel is generally implemented by providing an initial task to be executed and adding that task to the task queue. Compute nodes pick up tasks from the task queue and execute them. The execution of a task can generate more tasks, which are put in the task queue. These tasks operate on specific (key, value) pairs of WorldContainers, which can be looked up, processed and updated.

The "task" model is very efficient for computer clusters, mainly because it exposes a high amount of parallelism, by allowing any two independent tasks to be executed simultaneously on different compute nodes or even on the same compute node. Also, the "Futures" mechanism that the tasks are built upon makes sure that there is no blocking communication between any two compute nodes on the cluster.

However, there are also some disadvantages to using the "task" model:

- since it employs operations at a fine granularity level (i.e. user operations directly on (key,value) pairs in the WorldContainers, adding tasks to the tasq queue), the model is less portable than desired at the implementation layer (for example to a GPGPU architecture), and is somewhat biased to cluster-based parallelism.
- the difficulty of designing a performance model for kernels built on top of the "task" model, because the task is a very small unit of work (the execution of a kernel is made up of the execution of a very large number of tasks) and it is difficult to estimate how many tasks will be generated throughout the execution of a kernel
- the absence of partial execution determinism – two different runs on the same architecture could result in very different traversal of the call-graph associated to a kernel. This makes debugging more difficult, because it is hard to obtain any partial result reproducibility to help with debugging.

The alternative "MapReduce-like" model operates at a higher level of abstraction, the WorldContainer level. In this model the user will not work directly with existing (key, value) pairs from the WorldContainer, and will not add tasks to the task queue; rather, the user will specify bulk operations to be performed on WorldContainers. All programming primitives provided by the "MapReduce-like" model have a pure functional behavior (non-functional extensions will be discussed later): they take as input (parameters) one or more WorldContainers and a Functor (an instance of a class that contains a function that processes a (key, value) pair) and produce a new WorldContainer, containing the processed (key, value) pairs from the input containers. The atomic data element in the "MapReduce-like" model is the WorldContainer. This lends itself to designing more accurate performance models than the "task" abstraction did. The number of operations called on WorldContainers is much smaller than the number of tasks for equivalent implementations of the same kernel. In this way, by modelling performance for the few needed bulk operations one can model the performance of the kernels implemented on top of them.

The "MapReduce-like" model is inspired by Google's MapReduce, Roomy (a library for parallel disk-based computing implemented by Daniel Kunkle), the parallel loops in Lisp or Scheme and bulk database operations, such as join or select.

The advantages of the "MapReduce-like" model can be summarized as follows:

- The programming model provides a higher layer of abstraction for users and developers as compared to the "task" model. This leads to portable implementations, that can be used with no change on various differnt architectures (HPC clusters, GPGPUS, secondary memory clusters – such as clusters of machines with locally attached SSDs or disks).
- The coarser granularity of the operations that describe the "MapReduce-like" model allow for a more accurate performance modeling of the kernels.
- Using this model will correspond to a breadth-first exploration of the problem graph, which has some advantages for massively parallel jobs, probably the most important being a partial determinism in the computation. This determinism (reproducibility) is visible after data structure synchronization points. This is an improvement over the "task" model, in which the only type of guaranteed determinism was input-output determinism. An effect of the enhanced determinism in the computation is, among others, easier debugging.

2 Primitives of the MapReduce-like Model

This section presents the currently provided primitives in the "MapReduce-like" programming model. The primitives can be found in *madness/src/lib/world/worlddc.h*. Each primitive is accompanied by a code example. All examples can be found in *madness/src/lib/world/wfc_test.cc*. The currently provided "MapReduce-like" primitives are:

- map : maps a function to each `(key, value)` pair in the input `WorldContainer`. The function is given by the implementation of *call* in the class derived from `MapFunctor`.

  map:
  ```
  WorldContainer<KeyIn, ValueIn>,
  MapFunctor<KeyIn, KeyOut, ValueIn, ValueOut>*,
  WorldDCPmapInterface<KeyO>
  →
  WorldContainer<KeyOut, ValueOut>
  ```

  A `MapFunctor` is an instance of a class containing a member function:

  ```
  std::pair<keyOut,valueOut> call(const keyIn &k, const valueIn &vIn);
  ```

  The user must provide an implementation for the "call" function for each `MapFunctor` class used in a program.

  The "map" function is inspired from the "map" function in Lisp or Scheme, which operates on lists of elements and it is similar in behavior to it.

  Example:
  Given a `WorldContainer<int, int> dfc`, that contains the following entries: `(0,0), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7)` and a `SquareMapFunctor`, defined like:
  ```
  class SquareMapFunctor :  public MapFunctor<int, int, int, int>{
  public:
  ```

```
virtual std::pair<int,int> call(const int &k, const int &vIn){
return std::pair<int,int>(4*k, vIn*vIn);
}
};
```

one can use the following code:

```
SquareMapFunctor * sqf = new SquareMapFunctor;
WorldContainer<int,int> dfc1 = dfc.map(sqf, dfc.get_pmap());
dfc1.fence();
```

and `dfc1` will contain: `(0,0), (4,1), (8, 4), (12, 9), (16,16), (20,25), (24,36), (28,49)`

- mapToMany : maps a one-to-many function to each `(key, value)` pair in the input `WorldContainer`. The function is given by the implementation of "call" in the class derived from `MaptoManyFunctor`.

  mapToMany:
  `WorldContainer<KeyIn, ValueIn>,`
  `MapToManyFunctor<KeyIn, KeyOut, ValueIn, ValueOut>*,`
  `WorldDCPmapInterface<KeyOut>`
  $\rightarrow$
  `WorldContainer<KeyOut, ValueOut>`

  A `MapToManyFunctor` is an instance of a class containing a member function:

  ```
  std::vector<std::pair<keyOut,valueOut>> call(const keyIn &k, const valueIn &vIn);
  ```

  The "mapToMany" function is a generalization of the "map" function, allowing the user to map a `(key, value)` pair in the input to a variable number of `(key, value)` pairs in the output. The "call" function returns a vector of `(key, value)` pairs. This function is very useful in implementing graphs of problem dependencies, such as a recursive problem specification.

  Example:

  If `WorldContainer<int,int> dfc` contains `(0,0), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7)`, and we have a `MapToManyFunctor` defined like this:

  ```
  class SqThF : public MapToManyFunctor<int, int, int, int>{
  public:
  virtual std::vector<std::pair<int,int>> call(const int &k, const int &vIn){
  std::vector<std::pair<int, int>> out;
  out.push_back(std::pair<int,int>(4*k, vIn*vIn));
  out.push_back(std::pair<int,int>(6*k, vIn*vIn*vIn));
  return out;
  }
  };
  ```

  one can write this next piece of code:

  ```
  SqThF * thF = new SqThF;
  WorldContainer<int,int> dfc6 = dfc.mapToMany<int,int>(thF, dfc.get_pmap());
  ```

```
dfc6.fence();
```

And the result (`dfc6`) could be:
`(0,0), (4,1), (8, 4), (12, 9), (16,16), (20,25), (28,49), (6,1), (18, 27), (24,64), (30,125), (36,216), (42,342)`
But the result could also be: `(0,0), (4,1), (8, 4), (16,16), (20,25), (24,36), (28,49), (6,1), (12,8), (18, 27), (30,125), (36,216), (42,342)`
It is important to note that when the processing of two separate entries generates the same output key, but different output values, there is no guarantee as to which one of the output values will be present in the result.

- join2: joins two containers by the key produced from applying the join operator to each `(key, value)` pair in the first container. It is a generalization of the database "join" operation. The keys produced by the join operator when applied to some `(key, value)` pairs might not actually exist in the second container. The **bool** value in the result will be set to **false** in this case. Users should first check whether the value was set to **true** before accessing the key of the second container in the result container.

  join2:
  ```
  WorldContainer<KeyIn1, ValueIn1>,
  WorldContainer<KeyIn2, ValueIn2>,
  JoinOp<KeyIn2, KeyIn1, ValueIn1>*
  →
  WorldContainer<KeyIn2, std::pair< std::pair<KeyIn1, ValueIn1>, std::pair<ValueIn2, bool>
  > >
  ```

  A `JoinOp` is an instance of a class containing a member function:
  ```
  keyOut join(const keyIn& k, const valueIn& val);
  ```

  The value returned by "join" represents the join key for the input.

  It is possible that, for some entries in the first container, the `keyOut` returned by "join" is not present in the second container. In this case, the bool value from the corresponding entry in the result container is set to false, and the value of the `ValueIn2` data in the result is uninitialized (this might break if the entry needs to be serialized and sent to another compute node, because the serialization mechanism might not know how to deal with the uninitialized value).

  Example:

  Starting with a `WorldContainer<int,int> dfc1` that contains `(0,0), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7)`, a `WorldContainer<int,char> dfc2` that contains `(0,'A'), (1,'B'), (4,'C'), (9,'D')` and with a `JoinOp` defined like this:
  ```
  class IdJoin :  public JoinOp<int,int,int>{
  public:
  virtual int join (const int& k, const int& vIn){ return k;
  }
  };
  ```

  one could write this piece of code:
  ```
  IdJoin * id = new IdJoin;
  WorldContainer<int,std::pair<std::pair<int,int>,std::pair<char,bool>>> dfc3 = dfc1.join2(dfc2,
  ```

```
id);
dfc3.fence();
```

and the result would be: `(0,pair< pair<0,0>, pair<'A',true> >),`
`(1,pair< pair<1,1>, pair<'B',true> >),`
`(2,pair< pair<2,2>, pair <uninitialized,false> >),`
`(3,pair< pair<3,3>, pair<uninitialized,false> >),`
`(4,pair< pair<4,4>, pair<'C',true> >),`
`(5,pair< pair<5,5>, pair<uninitialized,false> >),`
`(6,pair< pair<6,6>, pair<uninitialized,false> >),`
`(7,pair< pair<7,7>, pair<uninitialized,false> >)`

- join3 : joins two containers by the key produced from applying the "join" operator to entries in the first container, but this time only include the boolean that specifies whether they key was found or not in the second container. Users should then use "join2" to access only the keys that exist in the second container.

  join3:
  ```
  WorldContainer<KeyIn1, ValueIn1>,
  WorldContainer<KeyIn2, ValueIn2>,
  JoinOp<KeyIn2, KeyIn1, ValueIn1>*
  →
  WorldContainer<KeyIn2, std::pair< std::pair<KeyIn1, ValueIn1>, bool> >
  ```

  A `JoinOp` is an instance of a class containing a member function:

  ```
  keyOut join(const keyIn& k, const valueIn& val);
  ```

  The value returned by "join" represents the join key for the input.

  The "join3" function should be used in the cases in which it is possible that a join key is not present in the second container. By calling this function, unlike "join2", only the **bool** value that describes whether the join key was present or not in the second container is attached to the entry in the result container, without the initialized/uninitialized value. This avoids the potential problems of "join2". Once the user has called "join3", they can now call "join2" only on the keys that the result of "join3" says are present in the result container to access the values of the `ValueIn2` data.

  Example:

  Starting with a `WorldContainer<int,int> dfc1` that contains `(0,0), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7)`, a `WorldContainer<int,char> dfc2` that contains `(0,'A'), (1,'B'), (4,'C'), (9,'D')` and with a `JoinOp` defined like this:

  ```
  class IdJoin :  public JoinOp<int,int,int>{
  public:
  virtual int join (const int& k, const int& vIn){ return k;
  }
  };
  ```

  one could write this piece of code:

```
IdJoin * id = new IdJoin;
WorldContainer<int,std::pair<std::pair<int,int>,std::pair<char,bool>>> dfc3 = dfc1.join2(dfc2,
id);
dfc3.fence();
```

and the result would be: `(0,pair< pair<0,0>, true >), (1,pair< pair<1,1>, true >), (2,pair< pair<2,2>, false >), (3,pair< pair<3,3>, false >), (4,pair< pair<4,4>, true >), (5,pair< pair<5,5>, false >), (6,pair< pair<6,6>, false >), (7,pair< pair<7,7>, false >)`

- filter: filter the container by the given predicate's call function boolean output.

  filter:
  ```
  WorldContainer<KeyIn, ValueIn>,
  Pred<KeyIn, ValueIn>*
  →
  WorldContainer<KeyIn, ValueIn>
  ```

  A `Pred` is an instance of a class that implements a function that returns a boolean (such a function is a predicate):

  ```
  bool call(const keyIn& k, const valueIn& v);
  ```

  This function is inspired from the "filter" function in Lisp or Scheme and it is similar in behavior to it.

- splitFilter : splits the container by the predicate's call function boolean output. The result is a pair containing two containers: the first has all the entries for which the predicate returned true, the second all the entries for which it returned false. It is very similar to "filter", but it also returns, in a separate container, all entries that did not match the predicate.

  filterSplit:
  ```
  WorldContainer<KeyIn, ValueIn>,
  Pred<KeyIn, ValueIn>*
  →
  std::pair< WorldContainer<KeyIn, ValueIn>, WorldContainer<KeyIn, ValueIn> >
  ```

- combine : computes the union of two containers. If the same key appears in both containers, then there is no guarantee of which entry is kept in the result.

  combine:
  ```
  WorldContainer<KeyIn, ValueIn>,
  WorldContainer<KeyIn, ValueIn>
  →
  WorldContainer<KeyIn, ValueIn>
  ```

- glo_size : computes the number of entries in a container

  glo_size:
  ```
  WorldContainer<KeyIn, ValueIn>
  →
  std::size_t
  ```

  It does not require an explicit fence (synchronization is built in).

- reduce : computes a summary of a WorldContainer.

  reduce<reduceOut>:
  WorldContainer<KeyIn, ValueIn>,
  LocalReduce<KeyIn, ValueIn, reduceOut>*,
  ParallelReduce<reduceOut>*,
  reduceOut*
  → void

  This is the only primitive in the "MapReduce-like" model so far that has side effects: it modifies the reduceOut parameters. A `LocalReduce` is an instance of a class that implements the function:

  ```
  void merge(const keyI& kI, const valueI& vI, reduceO * redInOut)
  ```

  A `ParallelReduce` is an instance of a class that implements the function:

  ```
  void merge(const reduceO * redIn, reduceO * redInOut);
  ```

  The "reduce" operation works by each compute node looping over the local entries of the container and applying LocalReduce::merge to the current key, value and `reduceO` buffer. This creates a summary of all the local entries in the buffer. Then all nodes communicate in a network tree pattern to aggregate the local summaries by merging them with ParallelReduce::merge. Node with rank 0 will hold the final summary in its local buffer after all communication finished.

- mapUpdate : map over a container and update a copy of another container. This is one of the most useful functions in this programming model and it combines multiple operations into one, functionally. It is a powerful primitive, since it maps over one container and it updates entries in another container based on user-specified criteria. As all other primitives in the "MapReduce-like" model except reduce, it has no side effects. An deep copy of the second container is created first, and all updates are sent to it, rather than to the original second container.

  mapUpdate:
  WorldContainer<KeyIn, ValueIn>,
  WorldContainer<KeyOut, ValueOut>,
  JoinOp<KeyOut, KeyIn, ValueIn>*,
  UpdateOp<KeyIn, ValueIn, KeyOut, ValueOut>*
  →
  WorldContainer<KeyOut, ValueOut>

  An `UpdateOp` is an instance of a class that implements the function:

  ```
  valueO call(const keyI& kI, const valueI& vI, const keyO& kO, const valueO* vO, const bool& present)
  ```

  The `present` parameter is true when there exists an entry `(kO, vO)` in the original second container and false otherwise. The pointer `value * vO` should be accessed only if `present = true`.

  The **mapUpdate** operation is inspired by the **map-update** functionality in **Roomy**. It was used to greatly simplify the implementation of the "MapReduce-like" "compress" kernel in "mra". Also, it is much more efficient that an equivalent implementation using only all other primitives.

3 Implementing MRA Kernels using the MapReduce-like Model

3.1 The "Reconstruct" Kernel

The "Reconstruct" kernel from "mra" is implemented this way:

Algorithm 1 The "Reconstruct" Kernel in MapReduce-like Style

Input: An initial `(key, tensor)` pair, which will be the initial element of a `tensors` container; the coefficients tree represented by container `coeffs`, which holds `(key, node)` pairs.

Output: The reconstructed `coeffs` tree, with coefficients pushed all the way down to the leaves.

1: **while** `tensors` is not empty **do**
2: `tensorsAndCoeffs` = join3 `tensors` with `coeffs` by same key
3: // the above operation aggregates data from containers `tensors` and `coeffs`
4: `processedTensorsCoeffs` = map over `tensorsAndCoeffs`: process all existing `(key, tensor)` pairs and create default tensors for non-existing keys
5: `tensors` = map to many over `processedTensorsCoeffs`: generate the many next level `(key, tensor)` pairs in the implicit problem graph
6: `justCoeffs` = map over `processedTensorsCoeffs` to extract `(key, node)` pairs
7: // next three operations are used to update `coeffs` with the `justCoeffs` entries
8: // collect all `(key, node)` pairs, both new and old:
9: allCoeffs = coeffs.combine(justCoeffs);
10: allTensorsAndNodes = join `allCoeffs` with `processedTensorsCoeffs` by same key
11: coeffs = map over `allTensorsAndNodes` to extract `(key, node)` pairs

The implementation of "reconstruct" in the task-based model is:

```
template <typename T, std::size_t NDIM>
Void FunctionImpl<T,NDIM>::reconstruct_op(const keyT& key, const tensorT& s) {
    1. PROFILE_MEMBER_FUNC(FunctionImpl);
    // Note that after application of an integral operator not all
    // siblings may be present so it is necessary to check existence
    // and if absent insert an empty leaf node.
    //
    // If summing the result of an integral operator (i.e., from
    // non-standard form) there will be significant scaling function
    // coefficients at all levels and possibly difference coefficients
    // in leaves, hence the tree may refine as a result.
    2. typename dcT::iterator it = coeffs.find(key).get();
    3. if (it == coeffs.end()) {
    4.     coeffs.replace(key,nodeT(tensorT(),false));
    5.     it = coeffs.find(key).get();
    6. }
    7. nodeT& node = it->second;

    // The integral operator will correctly connect interior nodes
    // to children but may leave interior nodes without coefficients
    // ... but they still need to sum down so just give them zeros
    8. if (node.has_children() && !node.has_coeff()) {
    9.    node.set_coeff(tensorT(cdata.v2k));
   10. }

   11. if (node.has_children() || node.has_coeff()) { // Must allow for inconsistent state from tra
   12.      tensorT d = node.coeff();
   13.      if (d.size() == 0) d = tensorT(cdata.v2k);
   14.      if (key.level() > 0) d(cdata.s0) += s; // -- note accumulate for NS summation
   15.      d = unfilter(d);
   16.      node.clear_coeff();
```

```
17.      node.set_has_children(true);
18.      for (KeyChildIterator<NDIM> kit(key); kit; ++kit) {
19.          const keyT& child = kit.key();
20.          tensorT ss = copy(d(child_patch(child)));
21.          PROFILE_BLOCK(recon_send);
22.          woT::task(coeffs.owner(child), &implT::reconstruct_op, child, ss);
23.      }
24.  }
25.  else {
26.      if (key.level()) node.set_coeff(copy(s));
27.      else node.set_coeff(s);
28.  }
29.  return None;
}
```

Note that the "task" model operates at the level of individual entries in the WorldContainer hashtables, whereas the "MapReduce-like" model deals with WorldContainers as somewhat "atomic" units. In the above descriptions of the "Reconstruct" kernel, line 2 from the "task" model is conceptually equivalent to line 1 in the "MapReduce-like" model (looking up an entry in a hashtable per task is equivalent to a join2 between two WorldContainers: one keeping the requests, the other one keeping the answers).

Next, line 3 of the algorithm in the "MapReduce-like" model processes the entries of the container obtained in line 1, and it is conceptually equivalent to lines 3–7, 8–10, 11, 16–17, 25–28 in the "task"-based model implementation.

Line 4 of the "MapReduce-like" implementation creates the `tasks` for the next level of the implicit problem graph and it is conceptually similar to lines 11–23 of the "task"-based model implementation.

The rest of the "MapReduce-like" implementation of "reconstruct" updates coeffs in a few steps. Using the primitive "mapUpdate", which was not available at the time "reconstruct" was implemented, one can merge all these last operations into one.

3.2 The "Compress" Kernel

The "Compress" kernel in "mra" was implemented in the "MapReduce-like" programming model in this way:

Algorithm 2 The "Compress" Kernel in MapReduce-like Style

Input: An initial `(key, key)` pair, which will be the initial element of a `keys` container, in which the second key is the parent of the first key; the coefficients tree represented by container `coeffs`, which holds `(key, node)` pairs.

Output: The compressed `coeffs` tree, in which the leaves have no coefficients, and all coefficients are in non-leaf nodes in the tree.

// Top-down scan of the task tree (and creating the tensors tree)

1: level ← 0;
2: **while** `keys` is not empty **do**
3: `keysAndCoeffs` = join2 `keys` with `coeffs` by the same key
4: `newkeys` = map to many over `keysAndCoeffs` to generate the next level of tasks, which holds keys and their parent keys
5: `tensors[level]` = map over `keysAndCoeffs` and, for each entry, create the corresponding node in the current level of a `tensors` tree; the node can have multiple children or it can be a leaf; the tensors tree must be explicitly created top-down
6: level ← level + 1
7: level ← level - 1
8: **while** 1 **do**
9: `tensorsAndCoeffs` = join2 `tensors[level]` with `coeffs` by the same key
10: **if** level ¿ 0 **then**
11: `upTensors` = mapUpdate over `tensorsAndCoeffs` to `tensors[level - 1]` (apply the equivalent of compress_op on the nodes at the current level of the tensors tree and update the parent nodes with the information of the current node after compressing the nodes)
12: `tensors[level - 1]` = upTensors
13: **else**
14: map over `tensorsAndCoeffs` to apply the equivalent of compress_op
15: `newCoeffs` = mapUpdate over `tensorsAndCoeffs` to `coeffs` to update the current level of `coeffs`
16: `coeffs` = `newCoeffs`
17: **if** level = 0 **then**
18: break from loop;
19: level ← level - 1

To model performance of these operations one must account for local access operations, local processing of entries in a WorldContainer and remote accessing of WorldContainer entries. Thus we obtain the following very high-level estimates for each operation, which will have to be expressed in a more concrete form if they are to be useful:

runtime(map) = runtime(iterating over local hash table) + size(local hash table) * runtime(functor operation) + runtime(all remote accesses for result) + runtime(synchronization)

runtime(join2 or join3) = runtime(iterating over local hash table) + size(local hash table) * runtime(find join key operation) + runtime(all remote accesses for join) + runtime(local accesses to result) + runtime(synchronization)

runtime(combine) = runtime(iterating over the two local hashtables) + runtime(iterating over the local result) + runtime(synchronization)

runtime(filter) = runtime(iterating over local hashtable) + runtime(iterating over result hashtable) + runtime(synchronization)

runtime(filterSplit) = runtime(iterating over local hashtable) + runtime(iterating over two result hashtables) + runtime(synchronization)

Using this programming model also introduces some determinism to the computation. After the synchronization of any WorldContainer, the data it contains should be the same in any instance of the program execution, on any architecture. This leads to easier debugging of the computation. Notice that in the performance estimates, the remote access time can be reduced by performing asynchronous access operations and

synchronizing the data structure as late as possible, to enable parallel execution of operations on multiple WorldContainers.