**MiniAMR – *The Good, The Add, and The Bugly***

*The Good:*

* Converting the original reference code to the MPI-**“like”** version that got named the “continuation cloning” implementation was good for several reasons:
  + It taught me miniAMR.
  + It taught me MPI (which I didn’t know before that).
  + It was a vehicle for developing “continuation cloning”, which then evolved into simply “cloning” because it turned out to be ideal for continuation cloning, fork cloning, and join cloning.
  + I demonstrated that an MPI-like version could be developed that did NOT suffer from the main drawback of MPI-Lite, which is that it violates the OCR application BKM of never stalling an EDT once it has launched.
* The reference code was then converted to “native OCR” application programming. This involved:
  + First “building the bones” of a series of continuation-cloning “root progenitor” EDTs that provide full-mesh services such as final collection and reporting of checksums and refinement topology (called “plot”), and final statistics reporting and shutdown. Note, importantly, that NONE of these centralized services ever require a global sync point. Rather, requests for these service percolate upwards, and since no reply is needed from them, the lower layers of the application can continue running. (This is distinctly different from how the reference code behaves! However, since most of these services are probably NOT considered part of the essential algorithm, but rather, are adjuncts to debugging and tuning, perhaps this major improvement isn’t really all that major at all.)
  + Then “layering on the muscles” in the form of continuation-cloning “parent” EDTs. Instances of these EDT sequences are created at the point that one block gets refined into eight (2x2x2), with one of these parent sequences providing partial checksums and plots and shutdown signals, and propagating them upwards, again with no reply, so this too avoids slowing the processing of the leaf mesh blocks. However, the parent also provide a service related to mesh unrefining, and this service requires it to return a response to all its 2x2x2 children. This service also has to propagate upwards, but the response comes directly from the parent, not an upward propagation of what is send up the line to grand-parents, etc., so this means that this case too should not cause any great big global syncs. (Note also that it is NOT necessary to propagate all the way to the root progenitor; the propagation stops one level short of that.)
  + Finally, “fleshing it out” with continuation-cloning EDT sequences of the leaf-level blocks. When a block refines, it is entirely replaced by a forking to a 2x2x2 set of finer blocks, plus an instance of a parent to provide services to those siblings. So a “block” only exists at the leaf. In reality, I implemented the “bones” and the “flesh” BEFORE I implemented the “muscles”, the latter only being necessary once I set my sights on implementing refinement.
  + Adaptive refinement was completed about a month ago; adaptive unrefinement was completed about a week ago. Both seem to work just fine – on a single X86. (See “bugly” for remaining issues.) While everything done up to refinement and unrefinement was guided carefully by what was done in the reference code, I found the structure of the reference code’s implementation of this feature, and its lack of commenting and meaningful program variables to be impenetrable. By this time, I had gained a thorough enough understanding of the algorithm that I decided to just implement the entirety of adaptive refinement and unrefinement from scratch. The steps to do both of these operations are far more tricky than one would initially guess, so there are very detailed comments in refine.c, where both of these mesh changes are implemented.
  + In accordance with my preferred coding style, the code is heavily commented, and I have not spared keystrokes in trying to make intelligible symbolic names. At some point, I started using pronouns such as I/me/my to refer to the EDT that is processing the very block for which the EDT was instantiated; he/him/his for that block’s “parent” (i.e. its direct service provider); and she/her/her for the neighbors of the “me” block. Sometimes, those neighbors are cardinal siblings (i.e. cardinally collocated neighbors of the same 2x2x2 block), cardinal non-siblings, just plain cardinal neighbors (comprising both siblings and non-siblings), and in a few places, just plain siblings (comprising all seven of the neighbor blocks for “my” 2x2x2 set). Note that sometimes, a neighbor isn’t merely a block at my same refinement level, but is sometimes a set of finer neighbors; and in the case of a non-sibling neighbor, “she” might be coarser, meaning the my face in her direction is merely one quarter of her face in my direction. Please be aware that while I started to use the pronouns, I did NOT do so throughout the code, and it would be prudent to consider either widening the practice or undoing, at the sensibility of whoever takes over the code.
  + While on the subject of comments, note that as I have passed over them many times, I have often found places where I said “refine”/”refinement” where I meant to say “unrefined”/”unrefinement”. I also sometimes found that I muddled the terms “finer” and “coarser”. While I think I’ve corrected most of these commentary mistakes, I was surprised at how often I made this dyslexia-like swap, so please bear it in mind as you read code and comments. \*\*\* Then correct carefully! \*\*\*

*The Add:*

* My main innovation on this work is the development of Cloning – Continuation Cloning, Forking Cloning, and Joining Cloning. I heartily recommend that this work should be embraced by others working on application development. Even so, I cannot claim to have gotten the feature to “product quality”, and it would benefit from being adopted by one adventurous soul who could put new eyes on the whole approach.

At its core, it allows “ocrifying” of applications without having to break the program structure of reference versions of the code at every point where an EDT has to be (or might have to be) cloned to a successor. For reference codes that are “flat”, i.e. that are NOT organized into nested function calls, “cloning” is easily accomplished by judicious use of a well-organized switch statement and some state data. My innovation goes beyond that, essentially allowing the switch cases to be (conceptually) able to span multiple levels of call topology. Moreover, it is “cleaned” up by the use of a half dozen macros that I wrote.

An important adjunct to the whole mechanism is that it preserves live variables from one instance of EDT to its clone (or clones, in the case of a forking clone operation). This is done by implementing a stack that, unlike the compiler-and-system-supported stack, is part of the state that is passed intact from an EDT to its clone. Even more significantly, this facility will auto-magically “fix” pointers in the predecessor EDT to their potentially different value in the successor (clone) EDT, so long as they are placed on the cloner-managed stack properly. This latter facility was used heavily when implementing the MPI-“like” implementation, where I did not at first know that most every pointer in the code as merely a base pointer, not an advancing cursor over, for example, rows, columns, or planes of an array. In the ocr-native variant, I’d pretty much recognized this characteristic of the code, and pretty much adhered to it, so it was not costly to recover base pointers from the guids of datablocks passed from one EDT to its clone, rather than attempting to keep “open” pointers from one EDT to the next. So I largely stopped using that part of the cloning innovation. But that does NOT mean that it should be jettisoned; it will still be valuable for other contexts where the pointers act (or might act) as cursors, advancing over their data.

* Nanny On Steroids: Another innovation is that I created the means to examine audit trails of the creation, passing, releasing, destruction of datablocks; the creation, passing, filling, and destruction of events; and the creation, equipping, execution, and destruction of EDTs.

This was done by replacing all ocr calls with similar calls to functions that begin with “gasket\_\_”. These functions write out an audit message (if –DNANNY\_ON\_STEROIDS is set in the makefile – and by the way, I often comment those sorts of –D’s by tacking on an “x”). The messages all begin with “NOS:” (Nanny On Steroids), and continue with “DBK:”, “EVT”, or “EDT”. The audit messages land in the terminal output, which I capture to the file “df”. (“df” stands for “day file”, which is what Control Data Corporation used to call what all the rest of us would have called a “log”.) Then, there are three shell scripts to manipulate them:

* + gen.sh breaks up all the audit messages into different files, rendering, for example, XXEVT\_<hex number> that contains all the audit messages pertinent to the lifetime of the EVT whose guid number is the <hex number>.
  + count.sh counts the number of times the EVT, EDT, or DBK is destroyed by one means or another. Usually, if not destroyed exactly once, something is wrong. The only exception I have recognized so far is that events that get recreated as a result of multiple lookups from a labeled guid range, are correct to have a single point of destruction for each life time. Rather than trying to “fix” this script to adjust for that, I just recognize this caveat is pretty much betrayed by the difference of that kind of EVT’s <hex number> compared to that of other events – and I just mentally discount it.
  + deldest.sh destroys all XX… files that have some form of destruction in them. If there are any XX… files left after doing this, it is because either the application failed (crashed, or you broke it out of a hang), or because you have a leak. In miniAMR, I have eliminated all the leaks. Note, though, that my prowess with scripts isn’t all that great, so this script does NOT validate that the destruction happens LAST, and ONCE, and ONLY ONCE. I also don’t have a script to analyze whether releases of datablocks happen correctly. They are mentioned in the audits, but not analyzed; so that is probably fertile ground for a great ToDo.

*The Bugly:*

* MiniAMR hangs when I attempt to run it on x86-MPI. I am NOT going to have time to dig into this. Likely culprits are failures to perform releases on datablocks the correct number of times and/or at the correct places in the code; declaring a datablock to be DB\_MODE\_RO when it is actually DB\_MODE\_RW; or perhaps failure to acquire a datablock before writing to it. See also the fourth bullet, below, as it might be the (or a) culprit.
* The refinement/unrefinement topology attained is different for the reference code than for the same test case run through the ocr-native code. (I never analyzed whether the MPI-“like” version suffers from this same malady, and I don’t know if Mark and Dave did this analysis for the MPI-Lite version.) This is recognized by comparing the “plot.###” files that come out of the application (when you select the –plot\_freq command-line option). These files are placed in …/install/x86. Be aware that the single-digit lines that appear in these files are the number of four-number-lines that follow. For the reference version, every MPI rank will disgorge its litany of blocks to the plot file as one big batch, and the size of that batch will be the single-digit number; but on the native-ocr version, every block is its own “rank”, so this single-digit number will ALWAYS be one. So, to compare the results, sort corresponding plot.### files for both versions, and then edit to remove all single-digit numbers. Then compare the two files that are now comprised of only the four-digit numbers. Those four-digit numbers are comprised of the refinement level of the block, followed by x, y, and z values for their “central” coordinate positions. It isn’t so necessary to understand the meanings of these coordinates as why the are different. My successor should plan on modifying both the reference version and the native-ocr version, dumping out old and new values of these block coordinates as blocks are refined, and again as they are unrefined. Sorting the results, the interesting part is to figure out where THE FIRST discrepancy arises, and then to dig into its cause.
* There are a few features of the reference version that are not yet implemented in the ocr-native version. Look for “FIXME” and “TODO” in the source code. An example of what’s missing: the reference code seems to have proxy code that corresponds to the original code perhaps applying multi-tap filters before exchanging halo data. Only ONE variant of the halo exchange code appears in the ocr-native version. If you pick one of the other halo-exchange methods on the command line, the code will throw an exception. (I do this, by the way, be executing “\*((int \*) 123) = 456;”. Works like a charm to crash the application. You look at the core dump in gdb, do a “bt”, and it takes you right to the point-of-crash, where I have documented the reason.
* The cadence of cloning is that at some point (such as when needing to receive a communication from one or more neighbors; or such as when needing to clone in order for a newly-created datablock to “ripen” and be available to write), the code has to “crawl-out”, create the successor, deck it out with its dependency slots (by supplying those datablocks that pass directly from this EDT to its clone; and by writing events that other EDTs will fill in with remote datablocks that they are supplying to the clone), and then “die”. The successor launches when all the events have been fulfilled, and it will “crawl-in” back to the point where the predecessor started the cloning operation.

The above description carries a bias, that I prefer the model of doing as much of the actual clone-decking work up at the top of the crawl-out chain, in a place and fashion that is hopefully common to many different points at which a cloning operation is needed. There are, for example, numerous places where “ReceivingACommunication” is the common reason for “throwing” a clone operation, and much of the decking-out of those cases is common.

Even so, there are differences as well. In unrefinement, there is a place where the only remote resource “I” need is a reply from my parent, whereas in halo exchanges, I need to receive faces and/or quarter-faces from various cardinal neighbors. This being the case, it is necessary to appropriately customize the usage cases.

I chose to create the concept of a “catalog” of dependencies that are being “equipped” by the current EDT for the needs of its successor EDT. But I did it WRONG! I decided that that depv array that an EDT receives upon entry could be refitted with the dependencies that EDT wants to pass on to its successor. Datablocks that want to pass along the full lineage of clones (e.g. “meta”, which is read-write, or “control” which is read-only), can do so by just staying parked in the depv array, and getting added as dependencies at the point where the successor EDT is instantiated and decked out with its dependencies. Other dependencies could be nullified (if received by this EDT but NOT to be passed to a successor), changed from null to something else (for something this EDT does NOT receive, but creates and passes to a successor), or overwritten (for a combination of both of the above).

The major flaw of this approach, is that there is an unpublished (or perhaps poorly documented) “feature” of ocr, in that when it starts an EDT up, it keeps a hidden copy of the pointer to the depv list, and when an EDT ends, it riffles though the list, performing ocrDbRelease on any guid in the list wasn’t released by the user code. In implementing that “feature”, the ocr authors did not anticipate that someone like me might overwrite those guids – and it causes a crash when I overwrite them with events! To “fix” that crash, I have to nullify all guids in the depv list after using them to “equip” the successor EDT, so that this “feature” of ocr doesn’t attempt to release guids of events. That works fine on x86, but this MIGHT be the cause of hanging on x86-mpi, which I am given to understand is less forgiving of failures to release datablocks.

There are two elements to a better fix. First, the ocr team should probably consider changing the API, such that an ocrEdtDep\_t should probably contain elements that are declared to be “const”, so that changing them like I did would raise an error.

Second, my successor should do something I no longer have the time to do: change the code that repurposes the depv array to code which instead populates a “successorCloneDependencyCatalog\_t”. The catalog should contain fields for guid and basePtr. The variable called “dbsize” of type array of DbSize\_t should become part of the catalog. That variable, though, is used inside the cloning macros and functions, and it will be incumbent upon my successor to figure out how to make it available to the cloning facilities in a general fashion while also keeping it usable for the specific needs of the different kinds of catalogs that need to exist (the rootProgenitor catalog, the parent catalog, and the block catalog). Then, this catalog should be embellished to include the following elements of meta-data that are not handled very elegantly right now: isDb is a boolean, that indicates if a catalog entry is a datablock (true) or else is an event (false) or NULL\_GUID (also false, and distinguished from event in the obvious way – the value in the guid element); isReleased is a Boolean that indicates whether the datablock has been released yet (and if it has not, then the gasket\_\_addDependence function should do so auto-magically, and it should nullivy the basePtr field); and accessMode (should be set to DB\_MODE\_RO, DB\_MODE\_RW, or DB\_MODE\_NULL as appropriate FOR THE NEEDS OF THE SUCCESSOR clone EDT, not the needs of the current EDT).

* And I suppose it might be “bugly” that the code has NOT been timed at all. That falls to my successor to do.
* The reference code has a ton of stuff related to gathering a “profile” of how the different MPI “ranks” are performing. There is no direct analog to doing that on native OCR. There is a very small amount of code in the native OCR version that started attempting to gather profile data, but when I realized that it was all out of whack, I commented out what I’d done, and stopped implementing any other parts of the reference code that dealt with profiling.

Potentially, my successor might want to try to profile what work gets performed on each XE (or “island”) of a real TG system. But to pull that off, you are going to need to get the ocr team to implement the feature I mentioned two or three months ago, that a dependence slot should be filled by that datablock from an array of datablocks that corresponds to the particular XE (or island) that ocr’s task scheduler selected for execution of the EDT. Then you can have a different profile accumulator for each XE (or island).