Unified Code Count Multithreading Notes

This contains some of the investigation / development work related to adding multithreaded operation to UCC to decrease running time on single or multiple CPU runtime hosts.

This is to give a high level view of various steps: analysis, design and development.
Of course additional details not found here can be found as inline or block comments at the end of various source files. When in doubt use Doxygen on the sources and/or read/run the code under the Editor/Debugger...

Contributor:
Randy Maxwell

Revised:
May 13, 2015 to June 2, 2015

Current efforts are with version **2013.04**
This is the latest released version for which I had access.

Times given here are ordinary (wall clock) times (to nearest second) and not CPU time used.
Benchmark tests use optimized Release builds.

References

UCC site
C++ source code and Release Notes and User Manual docs available for various versions
http://csse.usc.edu/ucc_wp/

UCC **forum**: Posted a note about adding Scala support (covered in another doc)

Sent mail
UnifiedCodeCount@gmail.com
about Compiler Warnings

Boost C++ cross platform libraries (Thread, IOStreams for Memory Mapped files)
Tested with 1.48.0 Follow install directions and compile the Thread library.
www.boost.org

Semaphore cross platform library
https://github.com/preshing/cpp11-on-multicore
Used just 1 source file **sema.h** and changed it to allow non C++ 2011 compiler to build.

Tools

C++ Microsoft Visual C++ 2010 Express to build/debug on my 4 year old Win7.1 laptop.
So I did NOT add any code that required C++ 2011 support or later.

Windows OS 7.x **Task Manager** (some screen shots below)
AMD **CodeAnalyst** to do Time based and some other Profiling

Background Investigation

Building:
Code defects found when compiling
When using the project settings as downloaded (Warning level 3) there are no warnings.
On projects I work with I **use W4 warning level** to let the compiler catch little things like:
Unreachable code, **assignment within conditionals**, etc.
When I rebuilt with W4 there are warnings about
potentially uninitialized variables (potentially real bad)
unreachable code
cast truncates constant value (bad if the compiler gets to choose implementation)

I think someone may have looked at the
conditional expression is constant
warning from such as:
while(true)
{ … }
blocks and decided that W3 was good enough.

I strongly recommend using W4.
The reason for this is that I feel really stupid when I spent time debugging something where if I had only turned on W4 warnings the compiler would have given a location and indication of the root of the problem.

UCC C++ Code Impressions/Feature Characteristics:
Overall structure and implementation seems clean and straight forward.
Design seems to be a console application with a **single thread** of execution.

User Feedback on **Large Jobs** (User Manual section 9.3):
"*Large jobs can be defined many ways. A large job may have large amounts of average size files, or a smaller amount of very large files, or files with very long lines of code. Computers with more memory, more disk space, and/or more processor speed will be able to process larger jobs using less time. This section will give strategies on how to do the best you can with what you have.*"

Then various strategies are given to reduce the amount of work that UCC must do.

Nothing is mentioned about having **more CPU cores as a way to speed up** UCC (assuming the run was partly Disk I/O bound).

Nothing is mentioned about moving the files to a Solid State Disk (or a RAM drive) to see if there is a speed up.

If the suspected problem area is UCC running out of memory during execution then perhaps there is one or more memory leaks. Found and fixed one for Debug builds.

Logical Source Lines of Code calculation (possibly inaccurate for some languages)
Ruby (and **Scala**) allows source lines to be continued on the next line without special continuation characters. This characteristic of Ruby is called out in the User Manual section **10.1** but I think better feedback would also **call out the possible inaccuracy in the output files for Ruby** as well. We know how sometimes developers get in a hurry to use a tool without reading the manuals. And the current approach may give a false sense of accuracy about Ruby LSLOC values.

Using UCC on UCC C++ source files:
Checking for most complexity

Performance Tests/Possible Code Changes

Nice to have:
Start, Elapsed and Stop Time recorded in the outfile_summary file.
(Implemented use of stdout (to console) to show times just before UCC exits)

I was going to make that little change myself and then run an unmodified test on about 200 Java files I have in various Java release directories to see the Elapsed time used before any changes.

I decided to NOT change the output format of files.
Instead current code will use stdout (console) to show time amounts of:
Total elapsed time: minutes and seconds
File list build time: seconds
File reading time: seconds
File analysis time: seconds

Goals:
Coordinate with UCC maintainers by submitting this as my plans.
(Done. Sent to the UCC mail address)

Do some performance tests with various code changes:
Use other C++ (or C) language features?
Yes. Added const modifiers on various procedure arguments to allow compiler to do more optimization.

Investigate/Design/Implement/Benchmark details of **multithreaded** changes:

Keep existing code/approach and have a command line flag
-threads 2 (or a higher integer) to specify the number of threads to use
to enable multithreading
With -threads >=2, have new code to do the multithreading.
Hopefully keep nearly all of the Language support files (classes) unchanged

After some testing, see if multithreaded changes are useful?

Then I was going to run again using some profiling tools to see if UCC is disk I/O bound.

If disk I/O bound then I would take a try to use multiple threads
(using portable PThreads maybe? No. Decided to use Boost Thread library.)

Code Survey

Existing Design/Implementation

```
int MainObject::MainProcess(int argc, char *argv[])
{
	/*
		Counting Process:
		1. Parse options
		2. Read input files
		3. Run counter on baseline files
		4. Identify duplicate files
		5. Print all results data
	*/
```

Step 2 **Read input files**
This reads in all the physical lines from all the files before step 3 is started

The problem with the above approach is that if there are many input files or if the input files are very large, there is a possibility of running out of memory. User Manual suggestions to avoid running out of memory seem somewhat weak see Large Jobs comments.

Design Alternate:
There is an intermediate step
1.5 Find wanted input files and put in a list

Then steps 1.5, 2 and 3 MAY not need to read in ALL the input files at once but rather
do

```
        1.5 Find an input file to process
        2   Read a found input file to a buffer of physical lines
        3   Find counters of interest from parsing the physical lines of the file
        Empty or reuse the buffer of physical lines
while there are more found files to do
```

Note that for a single threaded approach the above is not much faster than existing but the memory requirements are greatly reduced.

ALSO this approach allows to change to **multiple threads** to run in parallel on the files that may significantly reduce overall time to parse the files.

Possible Faster Duplicate File detection

See the User Manual section on the **-nodup** flag (9.2.2)
The order of given source lines between 2 files is not significant as procedures, classes, functions and so on may have been re arranged.

Also blank lines and comment lines are not compared for duplicate detection.

If the **-nocomplex** flag **is not used**
and
if the **-tdup** flag **is not used**

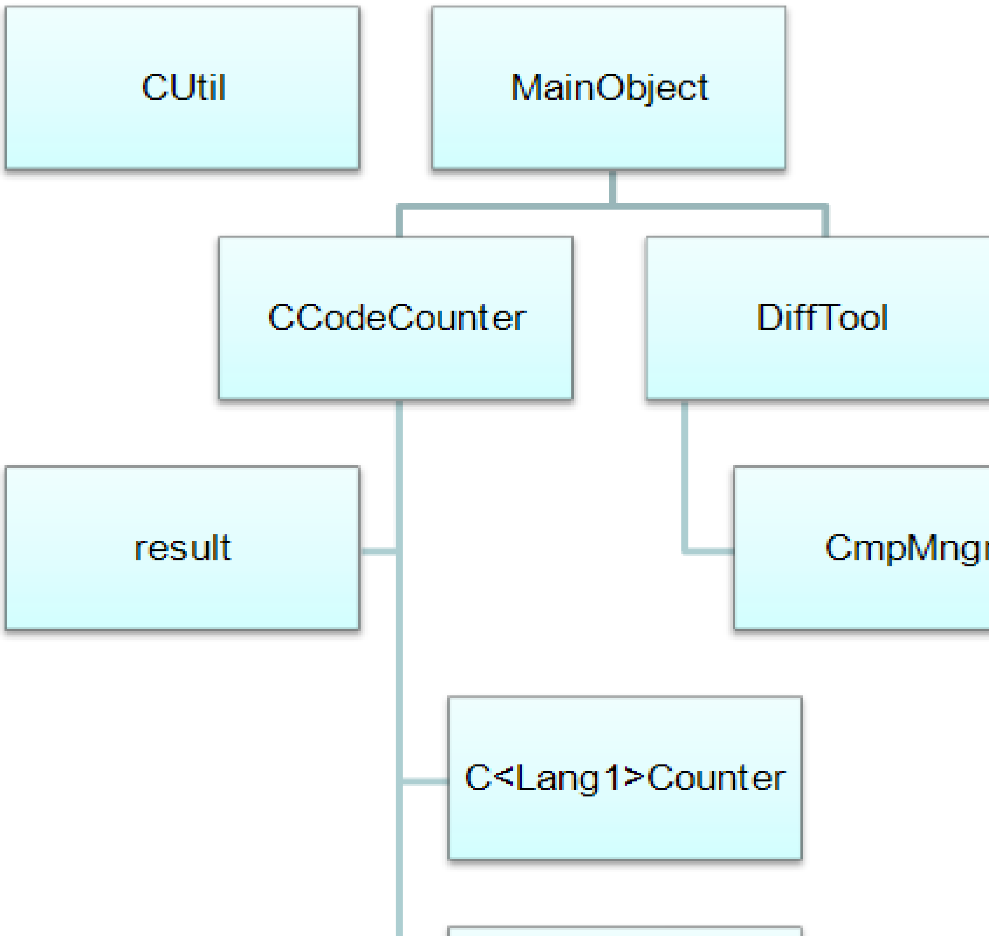
That leaves us all the good stuff to quickly determine if 2 files even CAN be duplicate:
2 files can be duplicates only if ALL the counts of the keywords and other complexity counts are the same between the files.

So detailed comparisons of 2 given files only need to be done if the counts are the same.
This could allow the program to quickly skip doing unnecessary detailed duplicate checking.

To Do: Investigate how checking for duplicate files works in more detail. Did that for better performance. Still needs to be done with this idea in mind...

Structure/Layout

Class organization and use of lower level utilities such as STL seems clean and straight forward. Below (single threaded) diagram is from: TOR-2010(3906)-72_Library.pdf



Note: The above diagram was done before the UI (Qt) was added.
And of course before the multithreaded changes described here.

C++ sources

UCC parsing UCC sources:

```
Cyclomatic Complexity
Total    Average  Risk
-----------------------------------------------
1075     31.62    High           MainObject.cpp
```

This is the overwhelming winner in the UCC source code Complexity Sweepstakes !
MainObject.cpp had both the highest total AND the highest average complexity.

MainObject.cpp file is TOO large at over **9,500** physical lines.
And it has **33 methods** in the class.

Refactored MainObject.cpp (do a simple repartition to some other source files)
Specific code to support various features of the different steps be moved to some helper files:

PrintUtil.cpp	has about **7,500+ lines** from MainObject
LangUtils.cpp .h	supports CounterForEachLanguage structure (Threads need this)
UCCGlobals.cpp .h	has global values that do not change after parsing cmd line args

MainObject.cpp now does:
Command Line Arg parsing
Start worker threads or enable faster Read/Analyze processing if wanted
Top level code that calls helpers: ReadAllFiles, ProcessSourceList, FindDuplicateFiles
Duplicate file checking
Output file processing
(threads are destroyed as part of MainObject destructor)

The above repartition also allows MainObject.cpp to create some helper work threads so that Duplication checking MAY run faster with parallel threads.
Parallel Duplicate checks not done yet.

DEBUG

(captured from Windows 7 console output window)
```
Reading source files.........................DONE
Performing files analysis and counting........100%  <<<
```

This somewhat misleading message shows BEFORE the first file's lines are parsed.
There was only 1 file to parse.

Simple Timing tests:
Running **DEBUG** build on UCC sources took **69 or 70 seconds** to finish.

Used both the **Windows Task Manager** Performance view
and the **Resource Monitor** CPU and Disk views (available at the same time)

During the run time the Disk activity sometimes hit 100% a few times
The CPU usage for the first CPU (1 of 2 available) showed close to 100% most of the time

So the **Debug** build with a single thread seems to be **CPU bound** rather than Disk I/O bound
This finding is counter to my prior speculation that UCC.exe was Disk I/O bound

Built a **Optimized for speed** version of UCC.exe and get the runtime and CPU and Disk I/O

Running **Optimized** build on UCC sources took only **1 second** to finish. **WOW ...**

Using **Windows Task Manager** Performance view I see a short spike on left CPU window to about 75% CPU utilization. So the optimization helped with overall CPU load as well (assuming there is not a measurement error due to the short duration of the test time).

Larger set of source files for testing
191 .java files plus various files of other types

Windows Task Manager Performance view
Test with **Optimized** build took only **15 seconds and CPU use was around 80%**

next few runs took only 7 seconds (Disk caching?) and CPU use was around 80%

Using Resource Monitor
CPU view
one CPU was 70 – 90% busy during the 7 seconds
other CPU was 30% or less (doing other processing not directly related to UCC)

Disk view
Disk I/O was 100% about half the time during the 7 seconds

MUCH Larger set of source files for testing
Linux version **3.11.6**
34,341 C source files plus various other files
Test with **Optimized** build took **1,884 seconds and CPU use was around 60% to 90%**
1,884 seconds = **31.4 minutes**
This is an average of **over 18.2 files processed per second**

Used the **Windows Task Manager** Performance view
UCC_Rel.exe allocated about **1.8 GB** of memory. At end of run allocation was **18 MB**
Size of UCC_Rel.exe file is 1.0 MB

When UCC_Rel.exe loads the allocation is only 412 KB

So it seems there were memory leaks during the Linux run

SMALLER part of Linux (just the fs – file system) directory tree
1,567 C source files plus various make files
Test with Optimized build took 42 seconds and CPU use was around 60% to 90%
This is an average of **over 37.3 files processed per second**

At end of run allocation was 2.90 MB

Example run:
UCC -dir "C:\Linux\Stable_3_11_6\linux-3.11.6\fs"
-outdir "C:\Utilities\TEST\UCC\Files_OUT" -ascii

```
Making list of source files...
```

```
Reading source files.........................DONE
Performing files analysis and counting.......DONE
Warning: Truncated 1 line(s) in file (C:\Linux\Stable_3_11_6\linux-
3.11.6\fs\cifs\netmisc.c)
Warning: Truncated 1 line(s) in file (C:\Linux\Stable_3_11_6\linux-3.11.6\fs\cifs\nterr.c)
Warning: Truncated 1 line(s) in file (C:\Linux\Stable_3_11_6\linux-
3.11.6\fs\cifs\smb2maperror.c)
Warning: Truncated 1 line(s) in file (C:\Linux\Stable_3_11_6\linux-
3.11.6\fs\compat_ioctl.c)
Warning: Truncated 3 line(s) in file (C:\Linux\Stable_3_11_6\linux-
3.11.6\fs\hfsplus\tables.c)
Generating results to files..................DONE
    Elapsed time (seconds) : 37
File list build time (seconds) : 0
  Files reading time (seconds) : 4
 Files analysis time (seconds) : 33
```

Although there were 1,567 .c files and some more make files analyzed, this is a marginal test case for expecting better results from using multiple threads because the overall runtime is less than 1 minute (optimized build using VS 2010 Express on my old laptop).

Analysis Supporting Multithreaded Changes

Step 0:
Find a use case where the runtime of UCC is over a minute for an optimized build.
Well I just decided to use the entire Linux distribution directory (version 3.11.6 in my case).

Linux 3.11.6 (stable release source file set)

```
Elapsed time: 27 minutes 1 seconds
File list build time (seconds) : 23
 Files  reading time (seconds) : 329   5 minutes 29 seconds
 Files analysis time (seconds) : 452   7 minutes 32 seconds
Find duplicates time (seconds) : 797  13 minutes 17 seconds
 Update  counts time (seconds) : 0
```

Design Decision: Use Boost Thread library
Downloaded/Installed/Built libraries for: versions **1.48.0**, 1.57.0, 1.58.0

Design Goal: Do not require anything special in the rest of UCC that is not needed for threads. **No new requirements for compilers. Hopefully Boost has good enough cross platform support.**

Added **UCCThread**.cpp and .h	Encapsulate implementation details of threads from UCC.
Added **LangUtils**.cpp and .h	Separate out some functionality to support threads.

Approach 1) File Read thread helper (develop proof of concept for worker threads):
Changed code in **MainObject**.cpp **ReadAllFiles**() to call a thread helper for each file to read.

Thread Design/Implementation Principle:
There is 1 access requirement; “**You can only Read what I Write. You can not change it.**” This is a simplification anyone doing multithreaded design should try to find as the behavior and implementation becomes more clear. (1 Writer, 1 or more Readers) Details are in the thread status structure. Threads use simple approach that both minimizes use of possibly scarce system resources like Semaphores, Mutexes, etc. while also allowing ease of development and future changes.

Benchmark testing example some stuff in a Java directory.

```
                                       LANGUAGE COUNT SUMMARY
                               Generated by UCC v.2013.04 on 5 2 2015
UCC -dir "C:\Program Files\Java" -outdir C:\TEST\UCC\Files_OUT -ascii

Language                      |   Number   |  Physical  |   Logical
Name                          |  of Files  |    SLOC    |    SLOC
------------------------------+------------+------------+-------------
Bash                          |          1 |          1 |            1
C_CPP                         |         17 |       7433 |         4161
CSS                           |          2 |       1029 |          826
Java                          |        191 |      24729 |        16950
JavaScript                    |         14 |       1545 |          954
Makefile                      |         10 |        265 |          208
SQL                           |         12 |       1545 |         1343
HTML                          |        133 |      11404 |         7833
XML                           |        467 |      13859 |         7997
------------------------------+------------+------------+-------------
Total                         |        847 |      61810 |        40273

Making list of source files...
Reading source files..........................DONE
Performing files analysis and counting........DONE
Warning: Truncated 1 line(s) in file (C:\Program Files\Java\jdk1.7.0\demo\db\pro
grams\vtis\data\DerbyJiraReport.xml)

Looking for duplicate files...
Generating results to files...................DONE
        Elapsed time (seconds) : 7                 <<<  No extra threads created
File list build time (seconds) : 1
 Files  reading time (seconds) : 0
 Files analysis time (seconds) : 3
Find duplicates time (seconds) : 3
 Update  counts time (seconds) : 0

==============================  THREADS  =========================

Started 2 threads.                                 <<<  2 threads created
Making list of source files...
Reading source files..........................DONE
Performing files analysis and counting........DONE
Warning: Truncated 1 line(s) in file (C:\Program Files\Java\jdk1.7.0\demo\db\pro
grams\vtis\data\DerbyJiraReport.xml)
```

```
Looking for duplicate files...
Generating results to files..................DONE
        Elapsed time (seconds) : 48
File list build time (seconds) : 0
 Files  reading time (seconds) : 42  <<< Much slower than 0 seconds (no extra threads)
 Files analysis time (seconds) : 3
Find duplicates time (seconds) : 3
 Update  counts time (seconds) : 0
```

Why are the 2 worker threads reading files so slow?
There is a **Semaphore** per thread to block the worker thread until the main thread has a file for reading. This prevents (supposedly) each thread polling while waiting for more work.

Lets run a longer test (**Linux fs** dir) for the Single threaded and with 2, 4, 6 worker threads.
Single thread (Task Mgr below)   **2** extra worker **Threads** (Task Mgr below)

```
        Elapsed time (seconds) : 38         Elapsed time: 1 minute 50 seconds
Build file list time (seconds) : 0    Build file list time (seconds) : 0
 Reading files  time (seconds) : 4     Reading files  time (seconds) : 76 << 1m 12s more
 Analyze files  time (seconds) : 29    Analyze files  time (seconds) : 29
Find duplicates time (seconds) : 4    Find duplicates time (seconds) : 4
 Update counts  time (seconds) : 0     Update counts  time (seconds) : 0

             4 Threads                            6 Threads
Reading files: 102 << 1m 38s more          Reading files: 127 << 2m 3s more
```

Adding more work threads with existing implementation clearly does not help.
Looking at **outfile_summary**.txt: there are **1,645** Files with **787,853** Physical lines.

So the original single threaded code in (approximately) 4 seconds does an average of:
411 Files read and **196,963** Physical lines processed each second.

I used a classic approach to (hopefully) speed up the “Read the files into memory” step:
“Divide and Conquer” partition the work to do among 2 or more work threads...
    “call a thread helper for each file to read”
For Java: **847 calls** to request 1 of the 2 available threads to open and read a given file.
For Linux fs: **1,645 calls** to request 1 of the 2 available threads to open and read a given file.

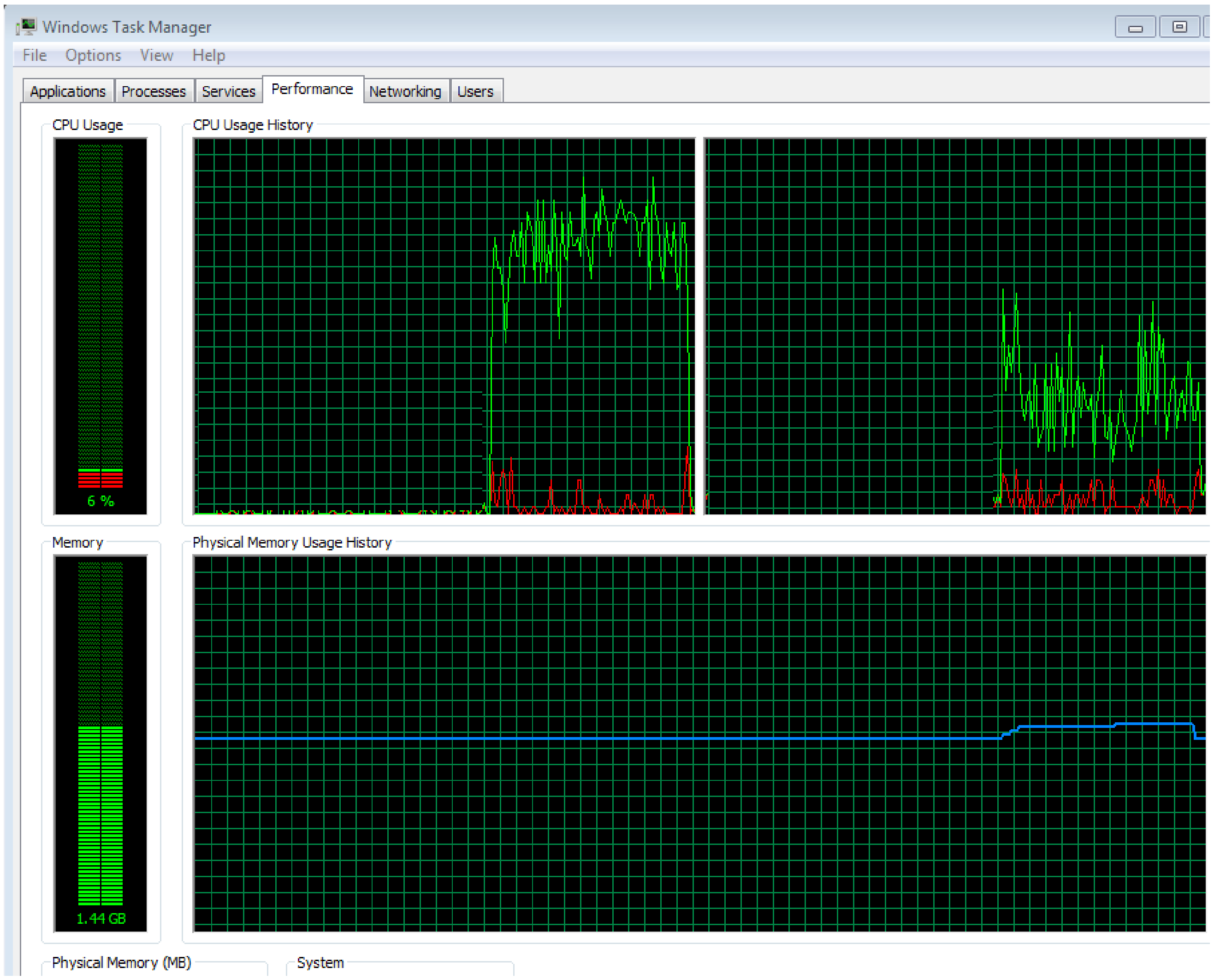
Consider the sequence of simplified processing steps (thread steps in parens):

Single thread	2 or more worker threads
For each file in the list of files	For each file in the list of files
	main thread calls to request worker to read a file
	(worker thread is unblocked and starts)
Open the file	(same code as single thread)
If open,	(same code as single thread)
read the Physical lines	(same code as single thread)
close the file	(same code as single thread)
handle Errors if any	(same code as single thread)
	(worker thread blocks waiting on a Semaphore)
Loop back to do another file	

Any blanks for the Single thread column take no time and are just formatting so that equivalent steps line up horizontally between Single and multithreaded. So this simple speculation about what is happening seems to support what the benchmarks found but let us think about some more details.

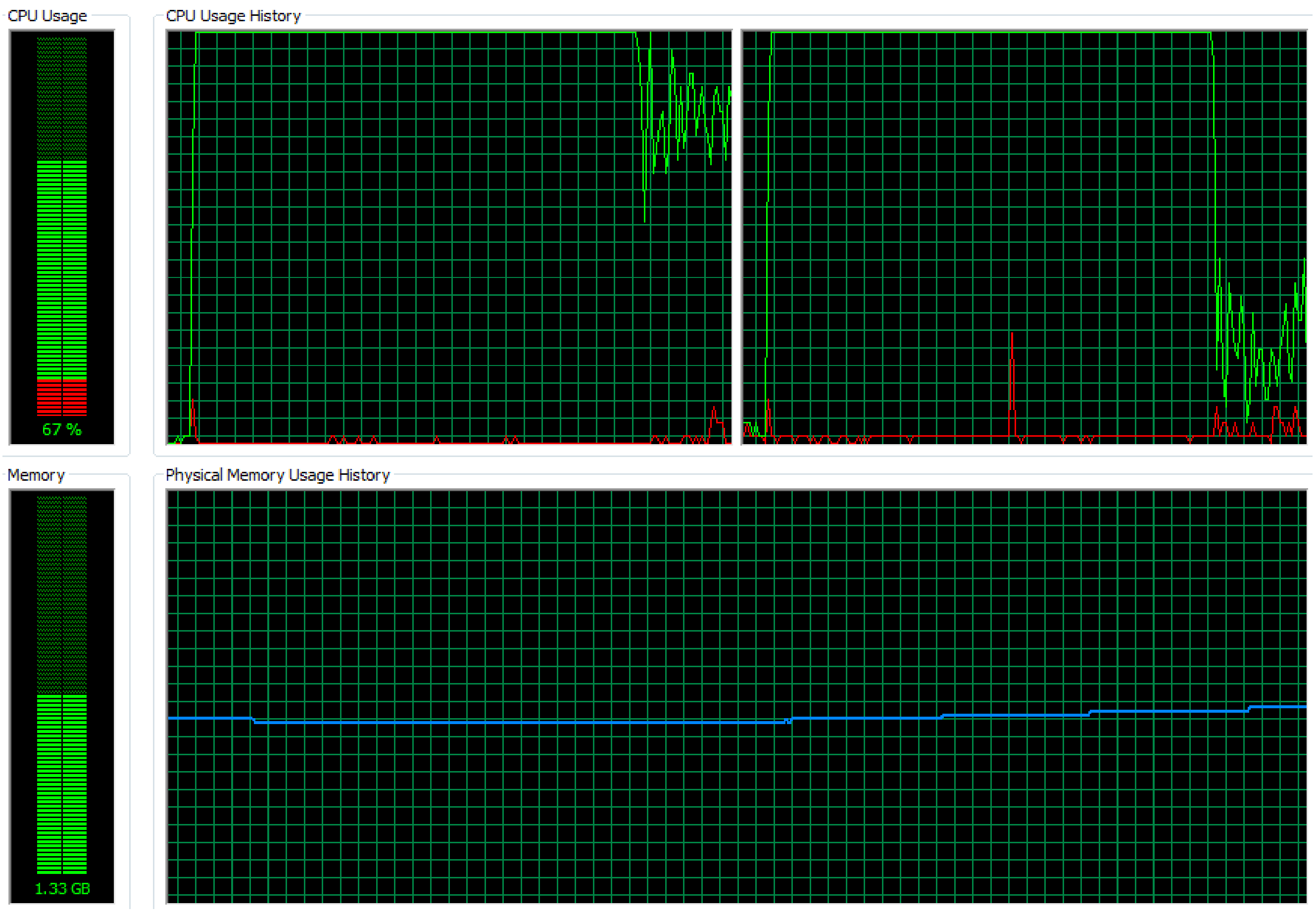
From an OS point of view:
a process is running with multiple threads and the main thread signals a Semaphore to wake another thread in the process.
The main thread may return immediately to get another file name to read or may yield() or sleep() – it doesn't matter that much as we understand this now what the main thread uses.
The OS thread scheduler (preemptive or cooperative) after some time will run the worker thread.

Lets use a tool to get more information about run time behavior.
For Windows I find that **Task Manager** helps me get the big picture.



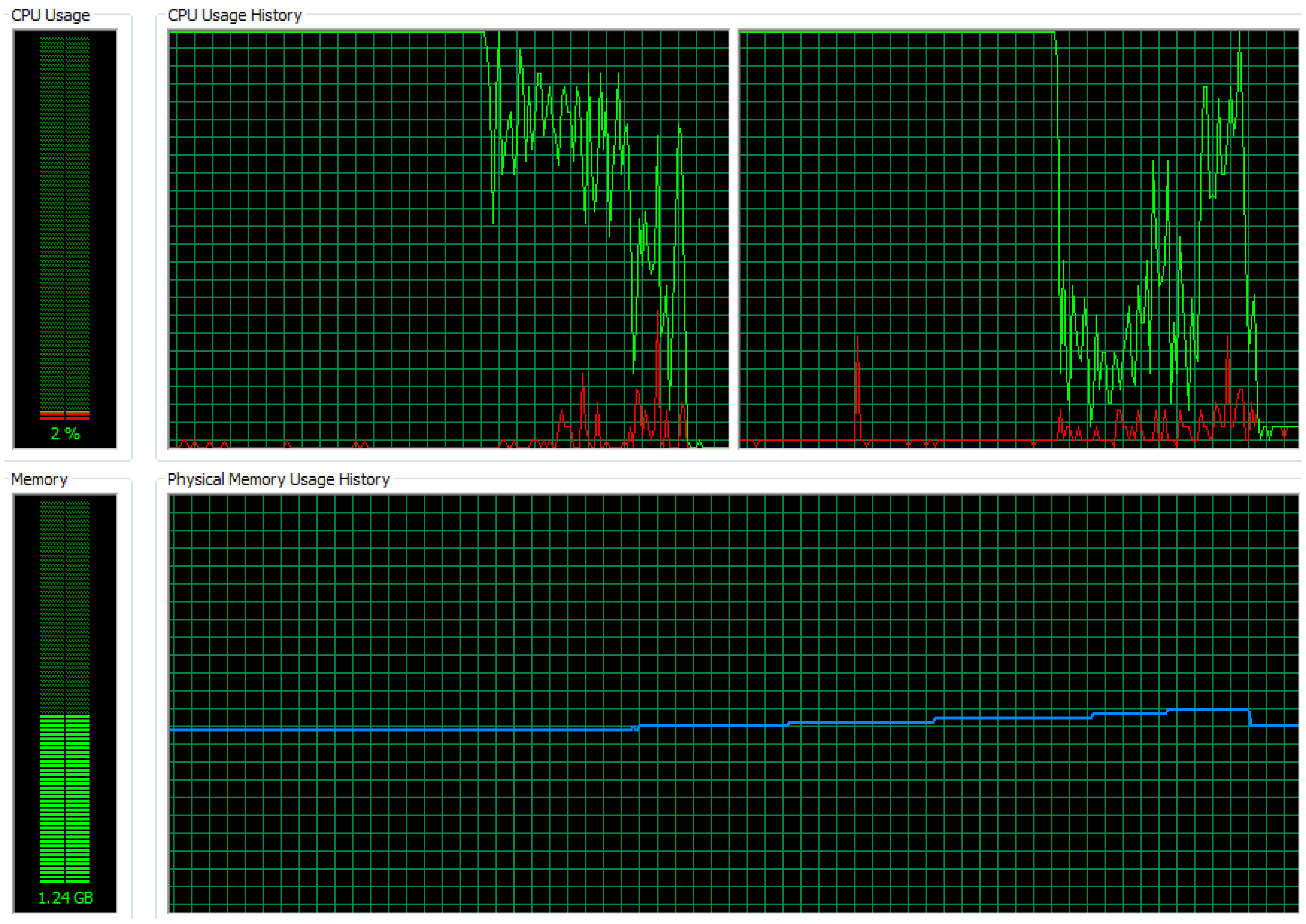
Single thread doing **Java** on 2 CPU laptop in **7** seconds.

UCC Processing Java dir in 7 seconds (UCC modified just for timing the steps)
UCC.exe (**single** thread) is on the CPU at the **top left** (about 65% to 85% usage)
The CPU window at top right is running other processes.
The **Green** line shows total program + **Red** line (kernel) CPU percent use.
The **Blue** line for memory aligns with UCC first allocating and then releasing memory at exit.
Just imagine the Blue line aligns with the window at left as it does for the right side.



2 worker threads starting doing **Linux fs** on 2 CPU laptop in **1 minute 50 seconds**.
The windows at the **top** shows both CPUs are running UCC.exe (at **100%** for **76** seconds)
The Green line shows total program + Red line (kernel) CPU percent use.
The Blue line for memory aligns with UCC allocating more memory.
I switched to **Linux fs** so this is not an entirely fair comparison but still gives an idea.

Aside from both CPUs being busy, the big difference is that during multithreaded file reading there is 100% CPU utilization but the overall useful throughput is LESS than single threaded. You can see the CPU utilization drops down when the rest of UCC goes back to single threaded execution to Analyze files, etc.



^
exit/free mem

For completeness, this shows the End of running 2 worker threads on **Linux fs** directory. Note that the top left window shows more CPU use on the right side as UCC goes back to single threaded operation. Blue line at right shows UCC.exe exiting and freeing memory.

Analysis of Multithreading results so far:

Pros:

We have reached the “proof of concept” stage as far as Thread coordination.
The threads can be created, blocked (sort of), given work to do in parallel and then give back results to be combined for downstream uses.

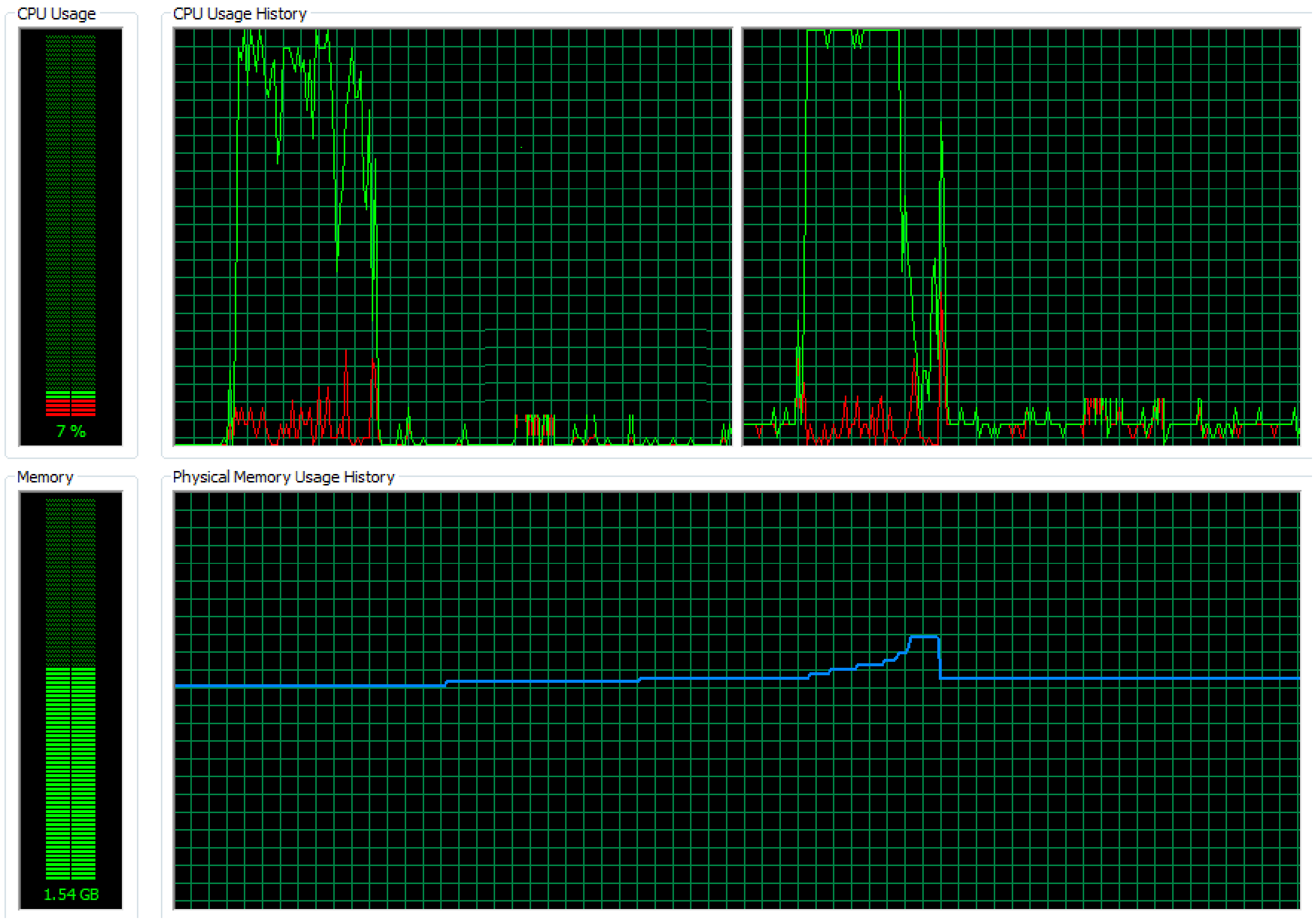
Cons:

Performance is WAY slower than the existing single threaded (unmodified) approach.

This is an example of one of the first lessons most developers find when changing code to be multithreaded... usually the first way you try to multithread some code or system will not give very impressive results. Then you must go back and challenge the design/implementation decisions and other tradeoffs from where you started to where your new code currently is running.

Rethink the code. What were the tradeoffs for the “proof of concept” version?
1) The main thread coordinates the 2 worker threads.
2) There was a very simple form to get a thread to work. Periodically it would wake up from a Sleep (or Yield) call and see if there was work available (another file to Read into memory. In other words Polling ! ! ! This shows as 100% CPU use even though 2 threads take MORE time to finish than the original single thread.
3) Another really big overhead that was not entirely clear to me until I went ahead and did the approach was that **1,645 calls to have a worker thread Read a file into memory is NOT a good way to partition the work.**

A better divide and conquer approach is to: **Divide the List** into a part for each thread to do. This greatly reduces the thread coordination overhead. Below is a more recent example.



2 worker threads doing **Linux fs** on 2 CPU laptop in **25 seconds**. **85 seconds** faster than early version of 2 threads and **now faster than single thread 38 seconds** by 13 seconds.
The windows at the **top** shows both CPUs are running UCC.exe (at less than **100%**)
The Green line shows total program + Red line (kernel) CPU percent use.
The Blue line for memory aligns with UCC gradually allocating memory and then exiting.

Lets look at some of the runtime details:

(**Linux fs** dir) for the Single threaded and with 2, 4, 6 worker threads.
Single thread (Task Mgr above)   **2** extra worker Threads (Task Mgr just above)

```
        Elapsed time (seconds) : 38          Elapsed time: (seconds) : 25 seconds
Build file list time (seconds) : 0    Build file list time (seconds) : 0
 Reading files  time (seconds) : 4    Reading & Analyze files   time : 19 vs 33 sec
 Analyze files  time (seconds) : 29
Find duplicates time (seconds) : 4    Find duplicates time (seconds) : 4
 Update counts  time (seconds) : 0     Update counts  time (seconds) : 0

           4 Threads                              6 Threads
Reading & Analyze files: 19           Reading & Analyze files: 18 (rounding?)
```

So on my 2 CPU core test HW adding more threads does not help over using 2 threads.
In fact adding threads use more CPU time without decreasing overall wall clock test time.

Careful reading of the times shows that individual times do not always add to total time.
Times are rounded to the nearest second. Lets say it took 26 seconds:
26 / 38 = 0.684 or uses **31 % less time** or if original is 100% speed, newer is **146% speed**.

3 Important Changes That Were Needed

1) **Do less work** to get work done. Dividing list of files was a critical insight in this case.
This snapshot of code divided the list of files to read among 2, 4, or 6 threads.
Which eliminated a large overhead of thread startup, do a little work, get back results cycles.
Now there are a few thread startup instances instead of over a thousand (1 per file).

2) **Use less time** to start a worker thread. Worker threads block on a cross platform Semaphore when not actually working. Main thread sets whatever info is needed by the worker to make a transition to a state wanted by the main thread and then signals the Semaphore. This completely eliminates polling by the worker threads. First try was to use the **Boost interprocess semaphore**. I was **running the debugger and traced into the implementation code of Boost** interprocess. There were calls to either yield() or sleep(). So when I thought I had avoided polling by using a semaphore, I really just had the polling going on in library code instead of application code. I am guessing that Boost code was fine for the intended use between processes but we needed a cross platform semaphore to use within the same process. Changed to use a different cross platform Semaphore library that properly blocked the thread without any polling (using native OS semaphores by the way).

3) **Push Down:** More useful processing is done by worker threads. After a file is Read into memory, the Analysis and Counting step is done. Then the next file in the list for the thread is Opened/Read into Memory/Closed (as far as the OS)/Analyzed & Counted and so on.
There is another Optimization available if the User does not want to check for Duplicates or Differencing then the memory holding the Physical lines and the Logical source lines are

released after the Analysis and Counting for a lower overall memory “footprint” at runtime.

Added another meaning to the -threads command line argument:
-threads 1
will allow a single thread to use the Optimized Read/Analyze approach as well.

So now the User can choose the “classic” UCC working as it did before with a single thread or experiment with Optimized approach for a single or multiple threads.

Other excitement before the 2 threads were working as above:

“**Shared Data**”
Original thread code was going to memory that the main thread had set up before the threads were started. Even though this was only READ access from the worker thread's code it had very poor performance and is termed “False sharing”. I refactored the thread code to make copies of what was really needed from the main thread and do some more set up during thread creation. So now each of the worker threads access almost exclusively the local stack as needed until all the files in the partial list given to each worker thread were processed. As part of when the worker thread finished the requested work, it would make available the results back to a communication structure that the main thread would use to update its own structures to continue processing as a single thread.

Unexpected Deadlock when starting worker threads
I was fortunate to catch an instance of Deadlock when debugging the app.
See the comment in UCCThread.cpp for details and resolution.

So have we reached a point of diminishing returns on improving multithreading in UCC?
No.
A later 2 Thread version not documented here does **Linux fs** in around **21 seconds**.

How much better multithreaded performance can be achieved (easy or otherwise)?
At a very rough guess... **5 to 50%** more
depends on some work not given here when I tried adding another language parser to UCC.

Details to follow sometime...

If you want to see some samples of using a **Profiler** to improve:
Differencing
`UCC_CA_Profile_DIFF_No_DUP_Details.txt`

or **Duplication** checking
`UCC_CA_Profile_DUP_No_DIFF_Details.txt`

Have fun!
Randy Maxwell