Some experiments with OpenMP® and LuaTEX

Parallelism & Concurrency

> “A system is said concurrent if it can support two or more actions in progress at the same time. A system is said to be parallel if it can support two or more actions executing simultaneously”

(Breshears, Clay (2009))

Explicit parallelism & concurrency

The user creates and manages the tasks.

Pros: Very likely we are always *over* 100% of speed of the previous *sequential* implementation. One can manage parallelism & concurrency, the last one being more flexible

Cons: Hard to understand, both at low level (libraries) and at abstract level (synchronization)

Some experiments with POSIX Threads and ZEROMQ™ have been shown at the 7h ConTEXt meeting.

Implicit parallelism

- 8 processors p_0, p_1,...p_7;
- `for(int i=0; i<8;i++){a[i]=2*i;}`
- each `a[i]=2*i` on processor p_i;
- Speedup $S = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{8}{1} = 8$ (800% !)

Implicit parallelism

This kind of parallelism is easy to understand and it's *implicit* in the `for` cycle. Theoretically is the perfect solution: if *all* the code can run in parallel the end user see a speedup almost equal the number of the processors (the underling OS still needs at least a processor too) *without* change anything.

Cons: we have *to modify the source code* of the application. Is it possible? Is it easy to translate a sequential algorithm into a parallel one? How much can we gain?

Implicit parallelism

Amdahl's law:

$$S = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{1}{(1 - F_e) + F_e/S_e}$$

T_{seq} execution time for the sequential version

T_{par} execution time for the parallel version of the program

F_e is the fraction of the original time in the sequential execution that can be converted in a parallel one

S_e the speed up that can be obtained if *all* the sequential code can be converted into a parallel one (John L. Hennessy and David A. Patterson (2012))

Implicit parallelism

8 processor: speed up max 100%.
To have a speed up $S = 4$ (i.e. 50% of the max speed up) is enough to rewrite 50% of the code ?
The Amdahl's law:

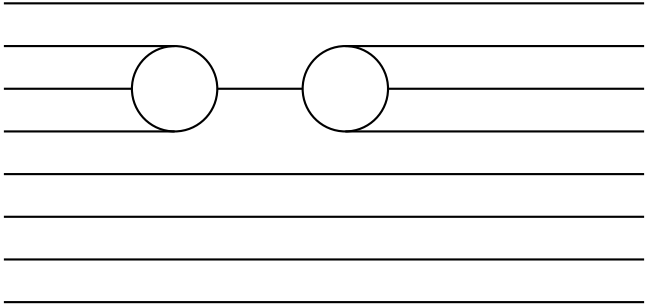
$$4 = \frac{1}{(1 - F_e) + F_e/8} \Longrightarrow F_e = 6/7 \approx 86\%$$

We have to rewrite 86% of the code: if we rewrite a 50% we have a speed up $S = 1.77$ i.e $\approx 22\%$ of the max speed up.

The Amdahl's law **is not linear on code !**

Implicit parallelism

Implicit parallelism is fragile: just a single (short) bottleneck degrades the performance:



Very likely we are always *under* 100% of max speed up.

Implicit parallelism: OpenMP®

> "The collection of compiler directives, library routines, and environment variables described in the document OpenMP 4.0.0 (available at [3]) collectively define the specification of the OpenMP Application Program Interface (OpenMP API) for shared-memory parallelism in C, C++ and Fortran programs."

There is a library, but it's *part* of the compiler suite: OpenMP® is compiler extension. Currently gcc & mingw support OpenMP® versions 3.0 as also clang for MAC OSX: Microsoft Visual C++ supports version 2.0 .

OpenMP®

The most important directive is `#pragma omp parallel` — but its effect can be unexpected: In the following code

```
foo_0();
#pragma omp parallel
  {
    foo_1(); foo_2();
    foo_3(); foo_4();
  }/* end parallel */
foo_5();
```

each `foo_`j(), $1 \leq j \leq 4$ is executed by *all* the threads available. This means that with N threads each `foo_`j() is called N times in random fashion for a total of $4N$ threads !
`foo_0` and `foo_5` are executed by one thread only.

OpenMP®

Instead the sections directive looks more natural:

```
#pragma omp parallel
  {
#pragma omp sections
   {
   ⋮
#pragma omp section
    {
     block_j
    }
   ⋮
   }/* end sections */
  }/* end parallel */
```

Each $block_j$ is executed concurrently but exactly once. This is the ``expected'' way: the execution inside each $block_j$ is sequential, all the threads have `}/* end sections */` as point of synchronization. Outside `#pragma omp parallel` the execution is sequential.

OpenMP®

Others directives are:

- `#pragma omp parallel for`
  `for(j=0; j<`$limit$`; j++){`
  `    `$block_j$
  `}`
  where each $block_j$ should be independent;
- `#pragma omp critical`
  `{`
  `    `*`block`*
  `}`
  that denotes a *critical region*.

OpenMP®

OpenMP® tries to be portable:

- if a compiler doesn't support a directive, the code is still valid and executed in sequential manner;
- already from version `2.0` there are enough directives (hence also Visual C++ is ok).

But this doesn't mean that these codes are equivalent:

```
foo_0();
#pragma omp parallel
  {
    foo_1(); foo_2();
    foo_3(); foo_4();
  }/* end parallel */
foo_5();
```

`foo_0()` {$4N$ threads} `foo_5()`
{$4N$ threads} at runtime!

```
foo_0();
/*#pragma omp parallel */
  {
    foo_1(); foo_2();
    foo_3(); foo_4();
  }/* end parallel */
foo_5();
```

`foo_0() foo_1()...foo_5()`
always!

LuaTEX

LuaTEX it's the merge of three interpreters:

1. Lua, with state L thread-safe;
2. MetaPost, with state M thread-safe;
3. TEX, with *no* explicit state T (so not even thread-safe).

state: the set of all the variables that describe the program;
thread: a sequential program (sub-process) executed by the OS concurrently with others threads. Threads *share* the same memory space — but the creation of a thread require 1/10 of the time to create a process;
destructive interference: interaction between threads where they write on a shared memory location in an unwanted manner;
thread-safe: a data structure that can be used with threads without destructive interference.

LuaTEX

Some problems with multi-threading & LuaTEX:

1. a state $\{L, M, T\}$ thread-safe means a potentially large rewriting of the code;
2. is there 'something' to translate from sequential to parallel? Is there some *theoretically* gain? If yes, is there some *concrete* gain?
3. how to use the threads? At TEX level? At C code level? What about multiple OS/platforms as in the current TEXLive distribution?

We need a starting point...

LuaTEX

Assumptions:

1. forget M and T: only L (thread-safe by design);
2. make a parallel version of `table.sort` (`table.psort`). The *theoretically* gain is known — we can measure the *concrete* gain;
3. we use OpenMP® — it looks the more portable among different choices.

LuaTEX `table.sort`

Lua implements the Quicksort algorithm in the recursive form:

1. given and an A select a *pivot* p and make a partition $A = \{L_A, p, G_A\}$ where $a < p \, \forall a \in L_A$;
2. sort L_A (recursive call);
3. sort G_A (recursive call);
4. the array A is sorted.

The sort is in place (good for memory) and *on average* it takes $n \log n$, n^2 in the worst case.

Can we make a parallel Quicksort ?

Yes, but ...

LuaTEX `table.psort` — Quicksort

But:

1. often the choice of the pivot results in L_A and G_A of very different size (unbalanced workload) and the solutions work *on average;*
2. recursive calls generate so many threads that the overhead of communication between them becomes predominant.

First conclusion: think to have as many threads as we want is a bad choice → there is a size of the sub-array (which ?) that triggers a sequential (i.e. *single thread*) sort.

LuaTEX `table.psort` — Mergesort

We change completely the point of view: we change the algorithm (Mergesort) and *set* the number of (virtual) processors (currently 4) and split the job in the following way:

1. subdivide the array A in four part of (almost) equal size: $A = \{A_0, A_1, A_2, A_3\}$;
2. run in parallel four threads ts_0, ts_1, ts_2, ts_3 so that ts_j sorts A_j using the (sequential) Quicksort;
3. run in parallel two threads tp_0, tp_1 where tp_0 does a *parallel merge* of $\{A_0, A_1\}$ and tp_1 does the same on $\{A_2, A_3\}$. At the end we have $A = \{B_0, B_1\}$, where each B_i is sorted;
4. run a final parallel merge of $\{B_0, B_1\}$, resulting in A sorted;
5. run in parallel four threads tc_0, tc_1, tc_2, tc_3 where each tc_j copies A_j on the final destination.

LuaTEX `table.psort` — Mergesort

The tricky point is the parallel merge with 4 threads: tp_0 uses two concurrent threads which in turn use a sequential merge — tp_1 does the same, so we have 4 threads.

The standard Mergesort needs an array — but Lua has only the *table*, i.e. dynamic and heterogeneous array. Every operations on a table require a state L (aka stack), so to avoid destructive interference we can :

1. create a unique $\{t_j, L_J\}$. This is the 'standard' solution. But how to share a table ?
2. lock/unlock the single state. This is possible but it's slow.

LuaTEX `table.psort` — Mergesort

Again, we choose another way:

1. disable the garbage collector just before the call to `psort` and enable it just after;
2. in single-thread mode, the state L (the *master state*) creates/destroys a pool of *auxiliaries stacks*; in multi-thread mode, each thread has one and only one stack (this is possible because the number of threads is fixed) and there is no destructive interference to access the pool;
3. each auxiliary table used is initialized with the correct size;
4. the user function for comparison is executed inside a critical region.

So: *one state and several threads!*

LuaTEX `table.psort` — Mergesort

Some results: 1/6

For the tests we have used an ASUS K55V from ASUSTeK COMPUTER INC. with single socket that hosts an Intel©CORETMi7-3610QM at 2.30 GHz with 4 cores and HTT, offering 8 virtual CPUs. The memory is 8GiB.

- *performance* $p = t_{\text{sort}}/t_{\text{psort}}$
- *efficiency* $e = \left(t_{\text{sort}}/t_{\text{psort}}\right)/(t_{\text{sort}}/(t_{\text{sort}}/4)) = p/4$

$p > 1$ and $e > 25\%$ means that `psort` is better than `sort`.

LuaTEX `table.psort` — Mergesort

Some results: several sizes 100 runs

Size	t_{sort}	t_{psort}	p	$e\%$
10M	15.7295	5.3158	2.96	74.00%
2M	2.8514	1.0064	2.83	70.75%
1M	1.3459	0.5250	2.56	64.00%
500k	0.6412	0.2725	2.35	58.75%
200k	0.2467	0.1169	2.11	52.75%
100k	0.1135	0.0662	1.71	42.75%
50k	0.0523	0.0400	1.31	32.75%
30k	0.0311	0.0291	1.07	26.75%
10k	0.0096	0.0184	0.52	13.00%
1k	0.0024	0.0136	0.18	4.50%

LuaTEX `table.psort` — Mergesort

Some results: English dictionary ≈ 100k, 5000 runs 3/6

Size	t_{sort}	t_{psort}	p	$e\%$
≈ 100k	0.177	0.104	1.70	42.5%

LuaTEX `table.psort` — Mergesort

Some results: several sizes, slow user function cmp

Size	t_{sort}	t_{psort}	p	$e\%$
1M	218.9900	92.6355	2.36	59.10%
100k	21.9988	9.6423	2.28	57.04%
10k	2.2980	1.0894	2.11	52.73%
1k	0.2311	0.1617	1.43	35.74%
100	0.0319	0.0351	0.91	22.73%

LuaTEX `table.psort` — Mergesort

Some results: several sizes, slow user function cmp 5/6
(pure) unprotected

Size	t_{sort}	t_{psort}	p	$e\%$
1M	221.1032	64.5599	3.42	85.62%
100k	21.6932	6.6300	3.27	81.80%
10k	2.1867	0.7104	3.08	76.95%
1k	0.2475	0.0851	2.91	72.70%
100	0.0322	0.0292	1.10	27.60%
10	0.0088	0.0211	0.42	10.43%
0	0.0046	0.0369	0.12	3.08%

LuaTEX `table.psort` — Mergesort

Some results: Lua lock/unlock enabled

Size	t_{sort}	t_{psort}	p	$e\%$
200k	0.3750	1.5608	0.24	6.00%

Maybe we can explain these figures: If we calculate the number of calls of the lock/unlock function, in this test there are $24\,353\,732 \times 2$ calls!

LuaTEX `table.psort` — Mergesort

Intel™ Cilk™ Plus: an alternative to OpenMP®, promoted by Intel™ together with Threading Building Blocks TBB.

Basically only two ``directives'': `cilk_spawn` and `cilk_sync`. The rest is OS dependent or given by TBB. It's oriented to C++ compiler, but it also works with the gcc C compiler (there is an experimental version for Linux).
Mergesort for Cilk™ Plus is well described in (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford (2009)) and (McCool, Michael and Reinders, James and Robison, Arch (2012)): in this case we don't set the number of cores, but we set the minimal size of the sub-array (1024 in the tests) that triggers a sequential sorter.

LuaTEX `table.psort` — Mergesort

Intel™ Cilk™ Plus with sizes power of 2

Size	t_{sort}	t_{psort}	$t_{\mathrm{sort}}/\mathrm{t}_{\mathrm{psort}}$	e
16 777 216	25.7886	10.5048	2.45	31%
8 388 608	12.4237	5.0129	2.48	31%
4 194 304	6.0006	2.5008	2.40	30%
1 048 576	1.3908	0.6699	2.08	26%
524 288	0.6619	0.3761	1.76	22%
262 144	0.3290	0.1752	1.88	23%
131 072	0.1626	0.0922	1.76	22%
65 536	0.0827	0.0589	1.40	18%
1024	0.0107	0.0113	0.94	12%

Note: efficiency e is calculated on 8 cores.

Conclusion

- a single Lua state a multiple thread seem to works
- performance is not as expected
- the parallel version can require a substantial amount of code

All this for a well designed library as is Lua ...what about TEX?

Short bibliography

1 Breshears, C. (2009). *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc.

2 Hennessy, J. L. and Patterson, D. A. (2012). *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann.

3 . http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf .

4 Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press.

5 McCool, M., Reinders, J. and Robison, A. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc..