Some experiments with OpenMP® and Lua\TeX
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 ConTÉXt meeting.
Implicit parallelism

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

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

- \( T_{seq} \) 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:

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

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 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

```c
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.
Instead the sections directive looks more natural:

```c
#pragma omp parallel
{
#pragma omp sections
{
  /* end sections */
}

#pragma omp section
{
  \( block_j \)
}

}/* 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 as point of synchronization. Outside

```c
#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:

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

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

foo_0() \( \{ 4N \ \text{threads} \} \) foo_5() \( \{ 4N \ \text{threads} \} \) at runtime!

foo_0() foo_1()…foo_5() always!
Lua\TeX

Lua\TeX 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.
Some problems with multi-threading & Lua\TeX: 
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.
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.
LuaT\textTeX	able.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 $t_{s0}, t_{s1}, t_{s2}, t_{s3}$ so that $t_{s_j}$ sorts $A_j$ using the (sequential) Quicksort;

3. run in parallel two threads $t_{p0}, t_{p1}$ where $t_{p0}$ does a \textit{parallel merge} of $\{A_0, A_1\}$ and $t_{p1}$ 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 $t_{c0}, t_{c1}, t_{c2}, t_{c3}$ where each $t_{c_j}$ 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.
Lua\TeX\ 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!*
Lua\TeX\ table.psort — Mergesort

Some results:

For the tests we have used an ASUS K55V from ASUSTeK COMPUTER INC. with single socket that hosts an Intel\textcopyright{}CORE\textregistered{}Mi7-3610QM at 2.30 GHz with 4 cores and HTT, offering 8 virtual CPUs. The memory is 8GiB.

- **performance** $p = \frac{t_{\text{sort}}}{t_{\text{psort}}}$

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

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

\textit{Bacho\TeX\ meeting 2014 — Bachotek OpenMP\textsuperscript{®} and Lua\TeX\}
LuaTEX table.psort — Mergesort

Some results: several sizes 100 runs

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

Some results: English dictionary \( \approx 100k \), 5000 runs

<table>
<thead>
<tr>
<th>Size</th>
<th>( t_{\text{sort}} )</th>
<th>( t_{\text{psort}} )</th>
<th>( p )</th>
<th>( e^% )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \approx 100k )</td>
<td>0.177</td>
<td>0.104</td>
<td>1.70</td>
<td>42.5%</td>
</tr>
</tbody>
</table>

BachoT\TeX meeting 2014 — Bachotek OpenMP® and LuaT\TeX
## LuaTeX table.psort — Mergesort

Some results: several sizes, slow user function cmp

<table>
<thead>
<tr>
<th>Size</th>
<th>( t_{\text{sort}} )</th>
<th>( t_{\text{psort}} )</th>
<th>( p )</th>
<th>( e% )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>218.9900</td>
<td>92.6355</td>
<td>2.36</td>
<td>59.10%</td>
</tr>
<tr>
<td>100k</td>
<td>21.9988</td>
<td>9.6423</td>
<td>2.28</td>
<td>57.04%</td>
</tr>
<tr>
<td>10k</td>
<td>2.2980</td>
<td>1.0894</td>
<td>2.11</td>
<td>52.73%</td>
</tr>
<tr>
<td>1k</td>
<td>0.2311</td>
<td>0.1617</td>
<td>1.43</td>
<td>35.74%</td>
</tr>
<tr>
<td>100</td>
<td>0.0319</td>
<td>0.0351</td>
<td>0.91</td>
<td>22.73%</td>
</tr>
</tbody>
</table>
Lua\TeX
table \textit{psort} — Mergesort

Some results: several sizes, slow user function \texttt{cmp} (pure) unprotected

<table>
<thead>
<tr>
<th>Size</th>
<th>$t_{\text{sort}}$</th>
<th>$t_{\text{psort}}$</th>
<th>$p$</th>
<th>$e%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>221.1032</td>
<td>64.5599</td>
<td>3.42</td>
<td>85.62%</td>
</tr>
<tr>
<td>100k</td>
<td>21.6932</td>
<td>6.6300</td>
<td>3.27</td>
<td>81.80%</td>
</tr>
<tr>
<td>10k</td>
<td>2.1867</td>
<td>0.7104</td>
<td>3.08</td>
<td>76.95%</td>
</tr>
<tr>
<td>1k</td>
<td>0.2475</td>
<td>0.0851</td>
<td>2.91</td>
<td>72.70%</td>
</tr>
<tr>
<td>100</td>
<td>0.0322</td>
<td>0.0292</td>
<td>1.10</td>
<td>27.60%</td>
</tr>
<tr>
<td>10</td>
<td>0.0088</td>
<td>0.0211</td>
<td>0.42</td>
<td>10.43%</td>
</tr>
<tr>
<td>0</td>
<td>0.0046</td>
<td>0.0369</td>
<td>0.12</td>
<td>3.08%</td>
</tr>
</tbody>
</table>
Some results: Lua lock/unlock enabled

<table>
<thead>
<tr>
<th>Size</th>
<th>$t_{\text{sort}}$</th>
<th>$t_{\text{psort}}$</th>
<th>$p$</th>
<th>$e%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>200k</td>
<td>0.3750</td>
<td>1.5608</td>
<td>0.24</td>
<td>6.00%</td>
</tr>
</tbody>
</table>

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.
# Lua\TeX\ table.psort — Mergesort

Intel™ Cilk™ Plus with sizes power of 2

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

**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


