Haglund / Hein
November 2014

DDS Performance and Benchmarking

Introduction

The key design criterion for DDS is solution speed. This seems nice and easy to quantify, but it turns out that it is not quite as one-dimensional in practice. This document outlines some of the considerations. Contributions to DDS are always welcome, and we will assess them on our somewhat subjective sense of "speed." At the moment, one meaning of "speed" is the result of running the supplied test program for the three main modes, i.e.

```
./dtest.exe masterDD.txt solve
./dtest.exe masterDD.txt play (the fastest)
./dtest.exe masterDD.txt calc (the slowest – takes hours)
```

This means that we mostly care about *average* times for *realistic* hands. We want to make sure that an answer is always given, even for pathological hands (see below), and we do care about improvements to worst-case performance, but mostly we care about average performance.

Solution modes

`solve`: The basic unit could be said to be a single hand with a declarer and a denomination/strain (NT or suit) given. This is indeed the best measure of "algorithmic base speed."

`calc`: One common use case is to solve a given distribution for all 20 combinations of declarer and strain, for instance in order to print it onto hand records. It turns out that partial results can be reused in order to speed up some analyses. It is quite easy within the structure of DDS to reuse results within a strain for different declarers – the transposition tables can be left in place. DDS makes no attempt to reuse anything between different strains as it doesn't fit with DDS's algorithms, but that's not to say that it can't be done.

In fact, the use case is probably not exactly as described. More likely 28 or 32 hands are to be solved for a complete hand record. So the speed of the entire task, as opposed to solving a single hand, is the relevant parameter. There is no particular reuse possible, but this gets to the topic of multi-threading and parellelism, see below.

Another common use case is to generate and analyze a lot of hands that fit certain criteria such as distribution and strengh, perhaps in order to assess whether to bid slam or not. DDS does not generate hands, but it does offer a batch mode which is useful in the context of multi-threading.

`play`: Yet another use, which is a recent addition to DDS, could be to judge the actual play of a hand relative to DD results. DDS offers a function for scoring each played card double-dummy. This could also be done by solving each position independently, but that is obviously very inefficient. Nonetheless, that is more or less the way that it was initially done in DDS v2.6. Subsequent reuse and stringent reduction of initialization code led to a dramatic improvement of a factor of 6-9!

Another use case, not currently offered by DDS, is to "walk a tree" of solutions, i.e. to be able play and undo individual cards played and to see the results. Programs such as Bridge Captain do this very nicely, but they do it by calling a DD solver repeatedly, and the solver may or may not reuse state

information. At least DDS now offers the option of solving a single "trace" through the tree, and it would be feasible to generalize this. On the other hand, DDS is so fast that these individual re-solves usually aren't visible to the user.

Thankfully changes tend to improve to worsen performance for all modes simultaneously.

Another use case is as an aid to bidding decisions or perhaps even bidding system design. We want to offer functions for this, e.g. "The bidding is at 3 spades and we want to know whether to invite 6 hearts or to stop in 4 hearts." This is in effect a generalization of the par functions. There are many ways that this could look, and we do have ideas. Inputs are welcome.

Parallelism and multi-threading

Modern processors have multiple cores that can be run somewhat independently in separate threads. When a batch of hands is to be solved, DDS can distribute the hands onto individual threads and re-integrate the results. To the user, the results is an apparent increase in speed, even though the "algorithmic" performance of an individual solve has not changed.

The improvement might be expected to be linear with the number of cores, but that is not really the case. The processors all need to access a single physical memory, and as the number of cores increases, memory bandwidth becomes the bottleneck.

Different algorithms can have different profiles in terms of memory bandwidth. For example, algorithms that "jump around randomly" a lot within memory incur more overhead. The memory sub-system of the computer generally has enough bandwidth for several threads, so as long as only a single thread is used, the penalty for this kind of algorithm is not noticeable. But when the performance is measured as a function of the number of threads, it shows up. One real-world example of this is the design of the transposition table. For a long time DDS had a tree structure, and it now uses flat lists. There is some overhead in memory size, but the speed has gone up. Thankfully this is the case even for a single thread.

Another effect of multi-threading is that some threads are certain to be done before others, leading to dead time and to a perceived loss of performance relative to the theoretical optimum. (At least the other threads have more memory bandwidth then…) This is related to the statistics of hand solve times. Specifically, some hands are much more computationally expensive than others. It turns out that the solve time for DDS is roughly Weibull-distributed, and the ratio between the longest and the average solve time within a batch can be quite large, 10 or more.

To be sure, this distribution is a purely empirical characteristic of the solve algorithm as it stands. But generally DD solvers use some kind of recursive pruning, and a lot of branching leads to a lot of nodes to be visited and therefore to a lot of run time. The number of number in fact correlates almost perfectly linearly with run time.

If there are lots of voids in a hand, this creates more options to consider, as that player has more playable cards on average.

If a suit is split AKQ – JT9 – 8765 – 432, it is also a lot easier on the solver, as there are only 4 actual cards to consider (AKQ are all equivalent, JT9 are all equivalent etc).

There are also differences between hands played in NT and in a suit. It turns out that in random hands, NT solves faster than the average suit. But among actual hands bid and played by real players, the average hand played in NT is more complex than the average hand played in a suit. Go figure.

Within a batch of, say, 200 hands, the hands are distributed onto, say, 4 cores. In fact they're not distributed evenly at the beginning – instead each thread reports back for duty when it is done with a hand, and it then gets the next one. If we're unlucky, one thread gets a very difficult hand as its last one, and all the other threads sit idle. This effect gets worse the fewer hands there are per thread, so large batch sizes are better.

One solution is scheduling, i.e. the algorithmic distribution of hands onto threads. If we could somehow determine ahead of time how long a hand would take to run, we could do the hardest hands first and we could split them evenly onto threads. DDS does include a scheduler as of v2.8, and the gains were meaningful. Nonetheless there is probably a lot of scope for improvement here.

For the record, the scheduler currently takes the following effects into account.

- Suit complexity, proposed by Bo Haglund, which is measured as the "fan-out" of a hand. KQT82.A43.97.T32 would have a fan-out of 4 + 2 + 2 + 2 = 10. A void is counted as the sum of all moves of the other suits in the hand. The sum of the four suits is the overall complexity. There is currently no difference implemented between NT and suit contracts.
- For play hands, the number of cards played. Human players tend to claim once the hand becomes uninteresting.
- Repetitions. For table hands this is easy (within a strain). But it turns out that some data sets also have the same cards repeated. For example play records of championships may have play from both tables. DDS does have a batch mode, but this requires precise alignment of the hands within the input. As of v2.8 DDS instead detects repeats within a batch of hands, even if the hands are not next to each other. (See "Simplicity" below.)

The strength split between the sides also plays a role, but the effect is not nearly as strong as for the fan-out, so this is currently not implemented (but the code for it is in the scheduler).

Simplicity

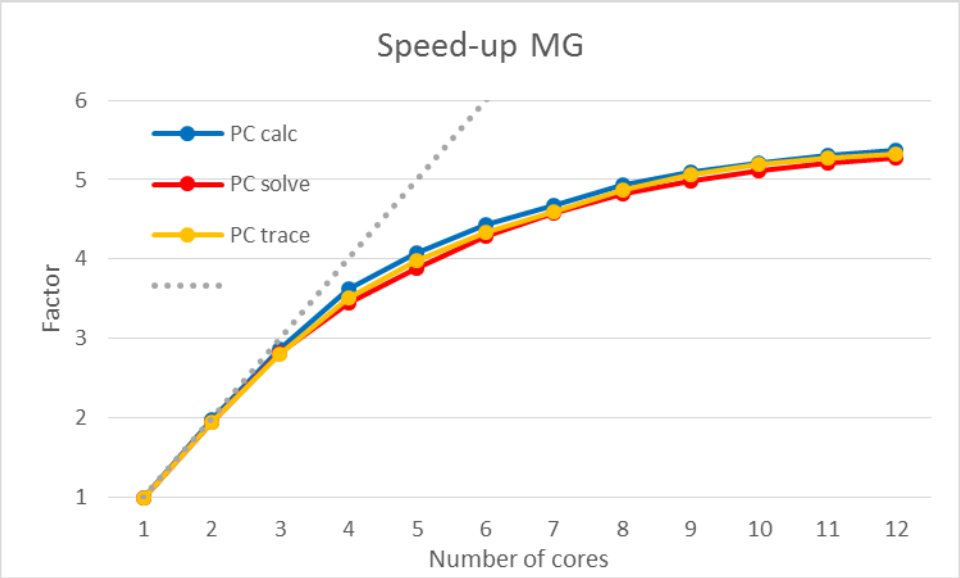
The number and range of DDS functions has increased over time. We don't have direct evidence of the use cases, but anecdotally it seems to us that not all users call the best functions for their applications. It's easy to "blame" the users for this, but the complexity and choice has increased, and programmers have other things to do in the own parts of the code, so it is understandable.

We want to eliminate some legacy functionality over the next versions (see the website).

The implementation of a repetition recognizer was another step in this direction. Probably the precisely aligned batch modes were too hard to use in practice or were not always noticed. Now DDS takes care of it behind the scenes. Is this a "real" improvement of hard algorithms. No. But it is worthwhile in practice nonetheless.

Results

The following examples are given for a specific 12-core PC. The results were generated as part of the development of v2.8 and they took a long time to generate, so it may be helpful to others to share them here.



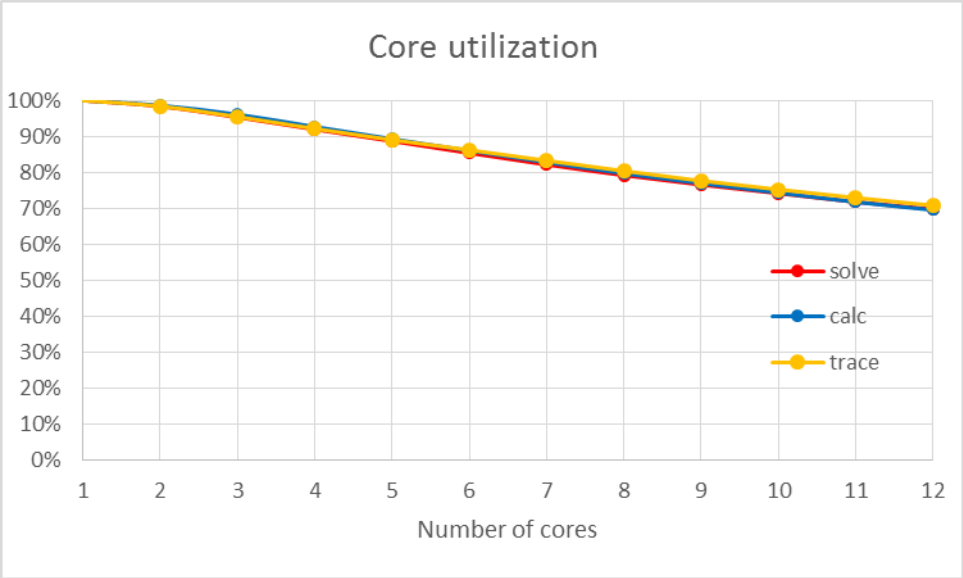
The above graph shows the relative performance of the three main solver modes as a function of the number of cores, for batch sizes of 200 for `solve` and `play` (shown as `trace` here) and 32 for `calc`. Historically `calc` had had a batch size of 10, which was way too little, but the DLL interface seemed limited in that regard. V2.8 seems to be OK, though.

- The scaling is close to linear up to 4 cores. That may not be the case on 4-core systems. More likely the 12-core system has a memory sub-system that is not enough for 12 cores running full blast, but enough for about 4 such cores at a given time.
- At the batch sizes given, the three main modes scale similarly.
- The performance flattens out at large core numbers.
- In earlier version of DDS the performance did not *seem* to scale as well. However it turned out that the data material (masterDD.txt) had repetitions of hands, and when run on a single core, DDS noticed this and optimized for it. On multiple cores it hardly ever happened that the same thread got both the repeated hands.

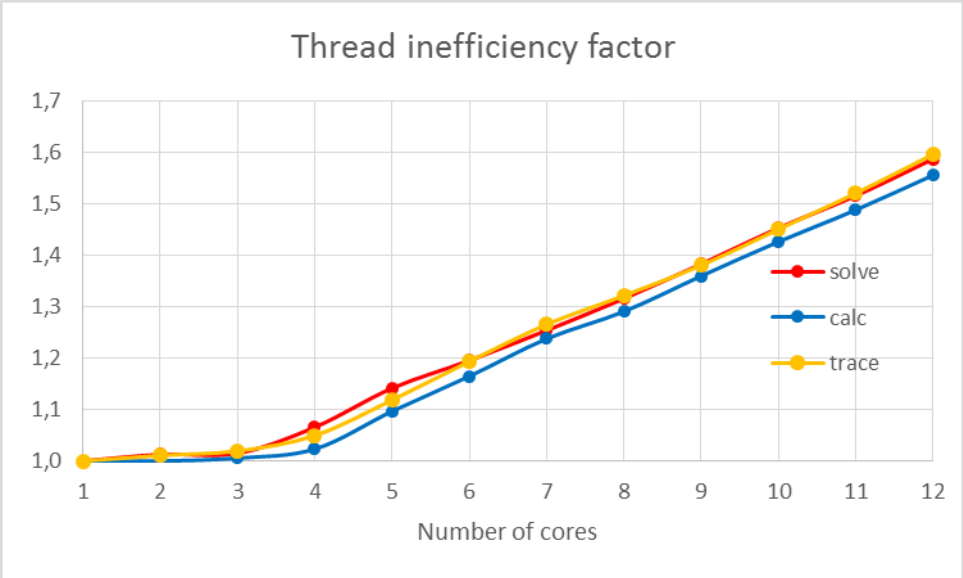
What exactly limits the performance? It turns out that the performance can be decomposed into (a) the performance of a single core and (b) the dead time created by multi-threading.

When a single core runs without interference from other cores, it has the entire memory sub-system to itself and so it runs fast. When it has to share the memory with others, it becomes slower, even though the algorithm hasn't changed at all.

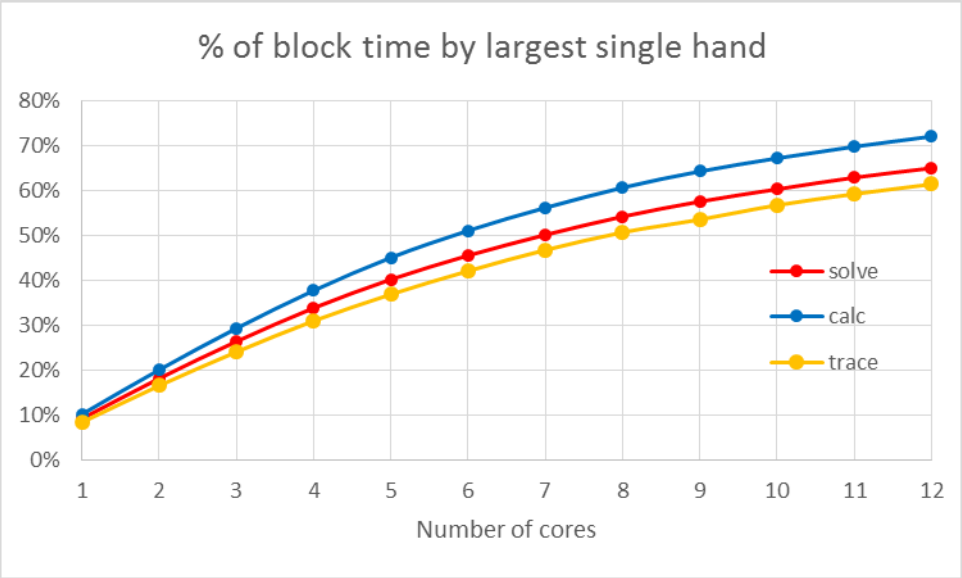
The following two graphs show this effect.



This is shown *on average* over the run time of a solve. So "12 cores" does not actually mean that 12 cores are running full blast next to each other all the times. Rather it means that up to 12 cores are available for DDS to use, with all the imperfections from scheduling etc. Still, we see that the memory bandwidth limitations are significant. The only thing we can theoretically do about that is to choose algorithms that use memory less often and more contiguously.



The above graph is the companion, showing how much of the time a thread is running. We see that up to about 4 threads things are pretty good. Each thread has about 50 hands in a batch of 200. At 12 cores, each batch only has about 16 hands, and considering the large statistical variation between hands, this is not enough to smooth out the run time. So this shows the theoretical potential of scheduling. If we could somehow schedule the hard hands first, this inefficiency factor would be closer to 1, i.e. not much loss. If we instead increased the batch side correspondingly, say from 200 to 600, then this would come back down to an extent, but the current DLL interface does have certain limitations.

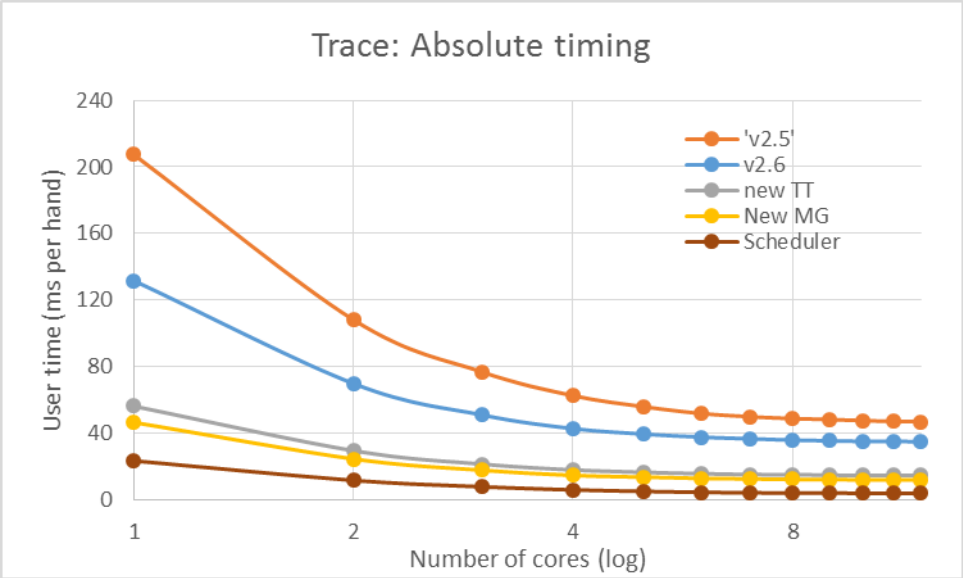


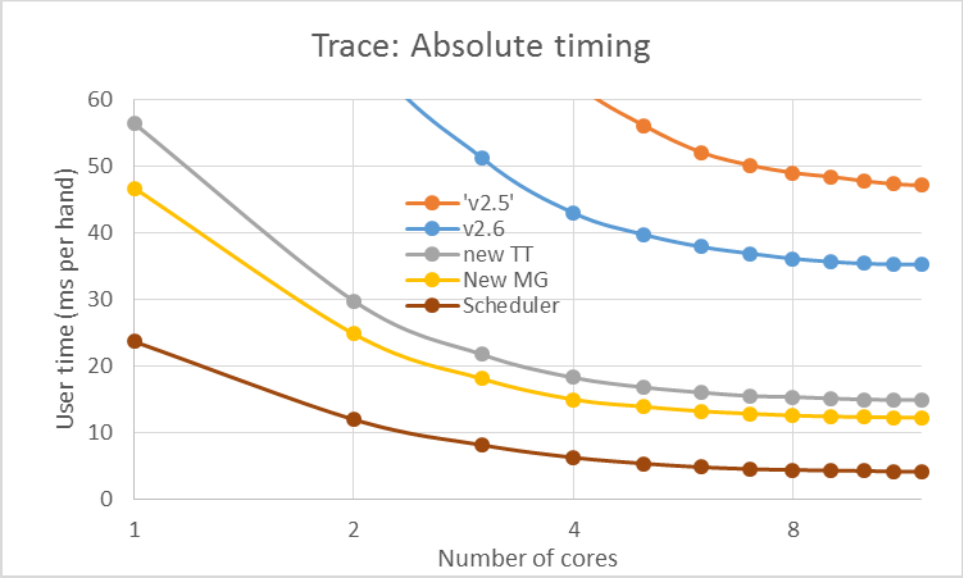
To continue along this line, the above graph shows how much of the overall invocation time is taken on average by a single solve. With 200 hands and 12 cores there are 16.7 hands per cores on average, so single hands are more important the more cores there are. The maximum of 200 hands is maybe 12 times more complex than the average single hand. For larger number of cores it becomes more important to cut down on the variation between hands, hence the scheduling.

Absolute results

Perhaps for historical interest, there are also partial benchmark results going back to about v2.5 (when Soren became involved). These are shown in the following pages for a number of cases. All cases are compiled using Microsoft CL on the same computer. The results are shown on two scales for each of three cases.

"v2.5" is close to the actual v2.5, but with the PlayAnalyser module added and without the initialization changes that were also made around that time. V2.6 is clear. "New TT" is the changes that led to v2.7. "New MG" is the new move generation that is included in v2.8. "Scheduler" is the scheduler which includes both scheduling and de-duplication and is also added in v2.8; these cannot be separated in the results – sorry about this. This is not a thesis ☺.





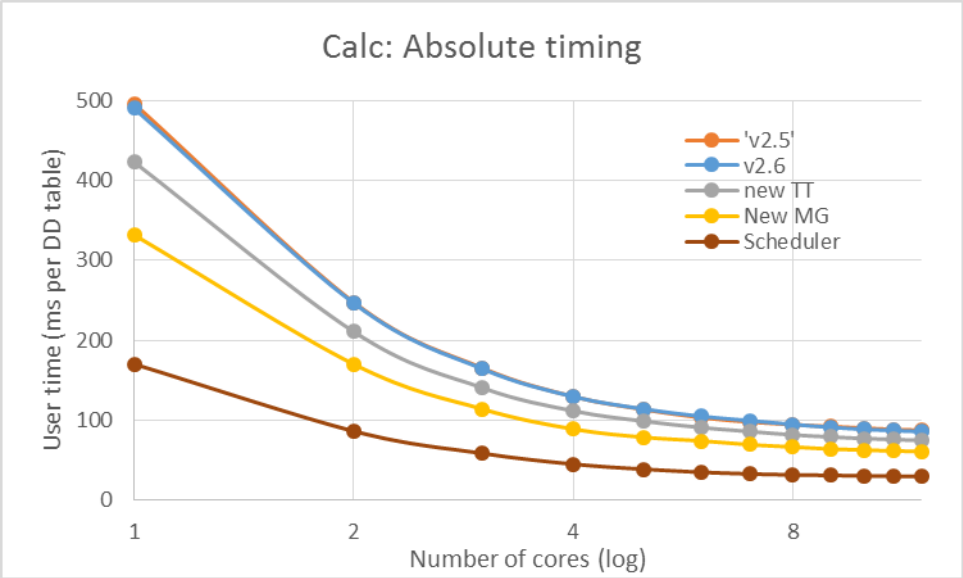
This is by far the largest absolute improvement of the benchmarks. "Trace/PlayAnalyser" was particularly sensitive to initialization, as this was done before each single card in a trace was analyzed. This is why the change between "v2.5" and v2.6 is only really visible for this benchmark. This improvement alone was worth 25-36%.

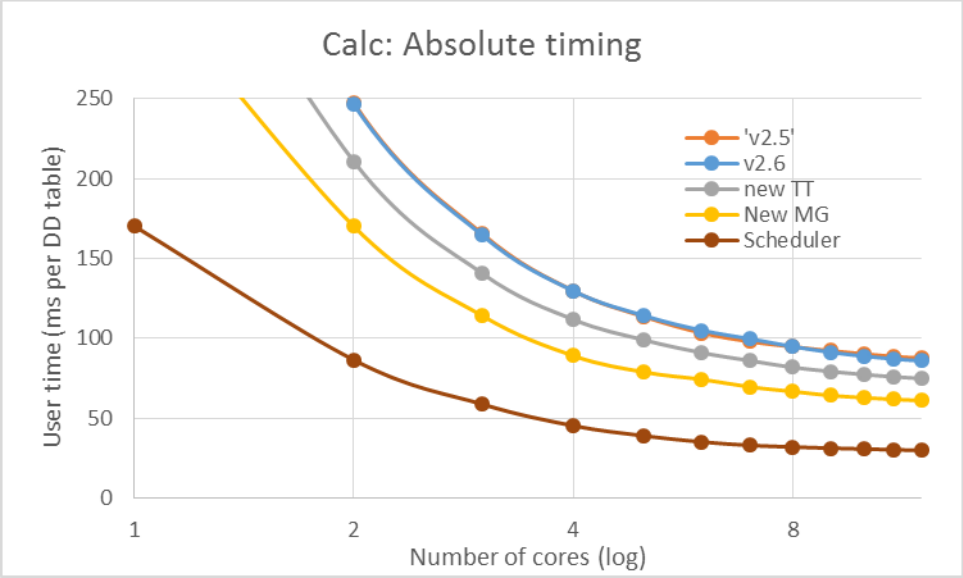
When making the TT improvements, there was also a change made to the way that the solver was invoked for each card which had a big impact.

The move generation had a normal impact, as on the other test cases.

And the de-duplication and scheduling as such also had a normal impact. But as part of this change, the iniatilization was improved even more, and this benchmark is particularly sensitive to initialization. And the benchmark at higher core numbers is also sensitive to the block size, which I increased to 200.

As a result, this function has improved overall, from "v2.5" to now, by a factor of 8.8 to 11.5.

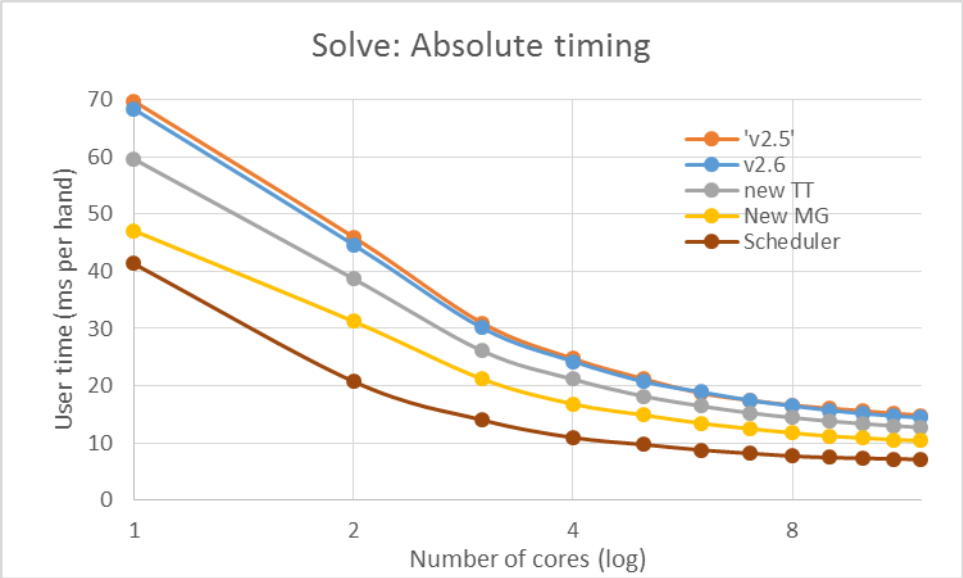


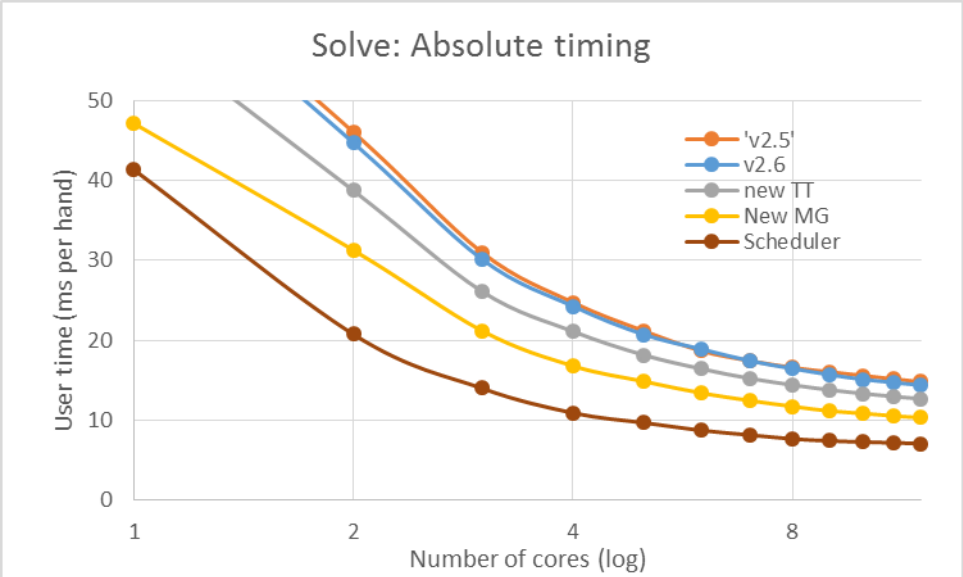


This is the benchmark where DD tables are solved in parallel. The initialization in v2.6 had almost no impact compared to all the work being done. The other changes are self-explanatory. As part of this, the block size was increased from 10 to 32.

The de-duplication is particularly useful here, as there was no mechanism in DDS for reusing results of identical hands in this mode so far. One might argue that the user should take care of it, but then users sometimes don't choose the best way to use DDS.

Overall, including gains from de-duplication (which are not in some sense "real" algorithmic gaains and are data-dependent), the gain is almost exactly a factor of 3.





This benchmark is harder to interpret, as there was/is an implicit reuse of at least some results with a single core. If the hands are sequentially next to each other in the file and if they have the same trump suit, then DDS would already reuse the TTs. Overall the gain (other than for a single core, where the gain with my data was a factor of 1.7) is in the range 2.1-2.2.