Increasing number of cores is not always beneficial

By David Livshin, published on October 22, 2018

It was observed that while applying parallelization by ***dco*** (see this) to the following code (only inner loop is shown, see this for details)

```
double a[ARRAY_DIM], c[ARRAY_DIM];
.
double wrap;
wrap = 0.;
for ( i = 0; i < ARRAY_DIM; i++ ) {
 c[i] = exp( wrap );
 wrap += a[i];
}
```

the resulting parallelized executable doesn't scale "as expected" when increasing the number of cores: using more cores don't lead to faster execution time. See this for another similar case.

In this article we will analyses this situation and provide the explanation for the observed behavior.

Denote:

LC to be a loop count (is equal to **ARRAY_DIM**)
TLC to be a loop count inside thread
NT to be number of threads (cores) utilized

Threads are numbered from 1 to ***NT*** and the loop count inside a thread is equal (approximately) to:

$$TLC = \frac{LC}{NT}$$

Inside every thread, the code generated by ***dco***, before entering main loop, calculates ***wrap*** to be equal to the sum of **a[i]** for all indexes **i** from 0 to the last loop index of the previously numbered thread. Thus number of memory loads ***NML*** necessary to establish this variable for the thread '***k***' is equal to

$$NML_k = (k-1)*TLC$$

Therefore the total number of loads executed outside the main loop is equal to

$$NML = \sum_{k=1}^{NT} NML_k = \sum_{k=1}^{NT}(k-1)*TLC = TLC*\sum_{k=1}^{NT}(k-1)$$

thus

$$\boldsymbol{NML} = \frac{\boldsymbol{LC}}{\boldsymbol{NT}} * \frac{\boldsymbol{NT} * (\boldsymbol{NT} + 1)}{2} = \frac{\boldsymbol{LC}}{2} * \boldsymbol{NT} + \frac{\boldsymbol{LC}}{2}$$

which clearly shows that the number of memory loads ***NML*** depends linearly on the number ***NT*** of threads (cores) utilized.

Conclusions

On the modern mult-core computer some resources are separate and independent from each other (e.g. execution cores) but other resources are still shared (e.g. memory). Therefore the care shall be taken not to abuse these shared resources. The example we analyzed will benefit from being rewritten using two-pass execution:

1. first path precomputes partial sums
2. the second path relies on the precomputed values to efficiently computed needed data

In the case that this is not done, the care shall be taken to establish optimal number of cores to be utilized as not to overwhelm shared memory system (possibly even giving up parallelization).