HPX backend for OpenCV
Summary for 2nd evaluation

List of my PRs:

- https://github.com/STEllAR-GROUP/hpx/pull/3335
- https://github.com/STEllAR-GROUP/hpx/pull/3214
- https://github.com/STEllAR-GROUP/hpx/pull/3365
- https://github.com/opencv/opencv/pull/11897

My repository with example applications and benchmarks:

- https://github.com/Jakub-Golinowski/opencv_hpx_backend

The image processing toolbox OpenCV supports multithreading in multiple ways, i.a. via TBB or OpenMP, but not with the use of HPX. Therefore, the goal of this project is to provide a reliable HPX parallel backend for OpenCV. As a result, current users of HPX will be able to seamlessly use OpenCV in their applications and OpenCV users will be provided with the possibility to utilize the HPX parallel backend.

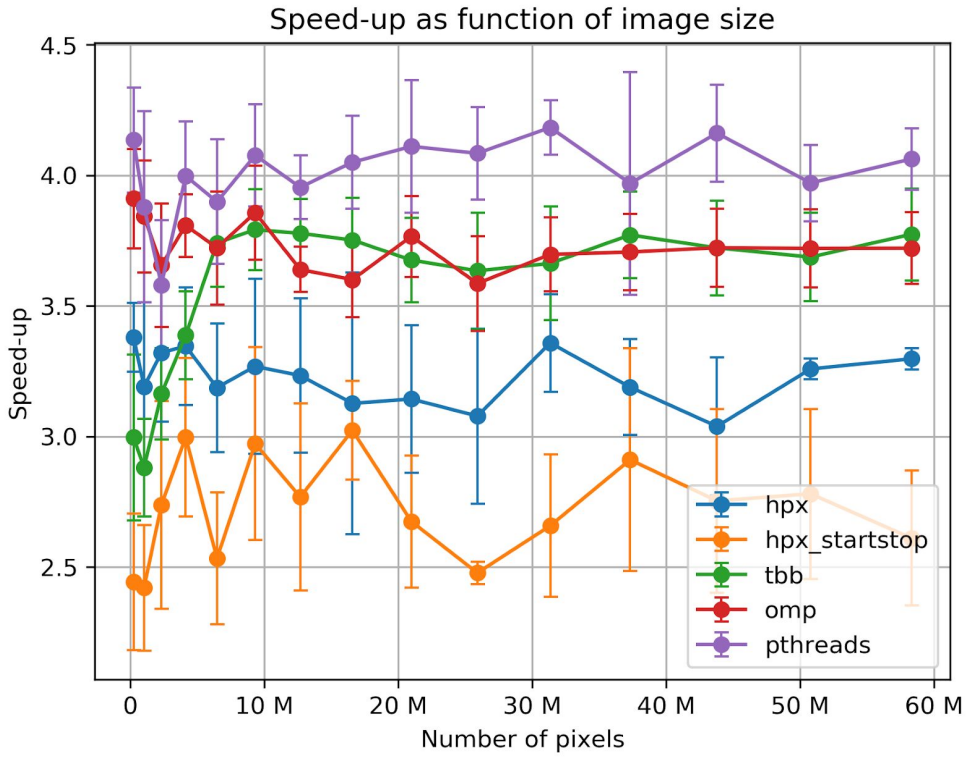
The OpenCV API contains a cv::parallel_for_() function which allows for the parallelization of simple for loops in the classic fork-join style. This function is both exposed to the end-user of the OpenCV library and used by its other functions. Depending on the OpenCV compile-time parameter the specific parallel backend is chosen to execute calls to cv::parallel_for_(). Apart from the backend implementation I added a cmake build option allowing to choose HPX as the parallel backend. However, it is important to note at this point that I developed *2 versions* of the HPX backend and the reasoning is presented below.

Since HPX is a general purpose C++ runtime system for parallel and distributed applications which allows user to build a DAG of his workflow instead of the classic fork-join approach, the primary use-case of the backend we have in mind is when a user of the HPX library wants to employ functionality of OpenCV in his application as part of the above mentioned workflow DAG. From now on I will refer to this backend version as primary.

In order to achieve the functionality of the primary backend version described above I had to implement it in such a way that it assumes that the HPX runtime was started by the user before the the call to the cv::parallel_for_(). Only in this way the full flexibility of workflow DAG construction and runtime lifetime can be preserved in user's hand.

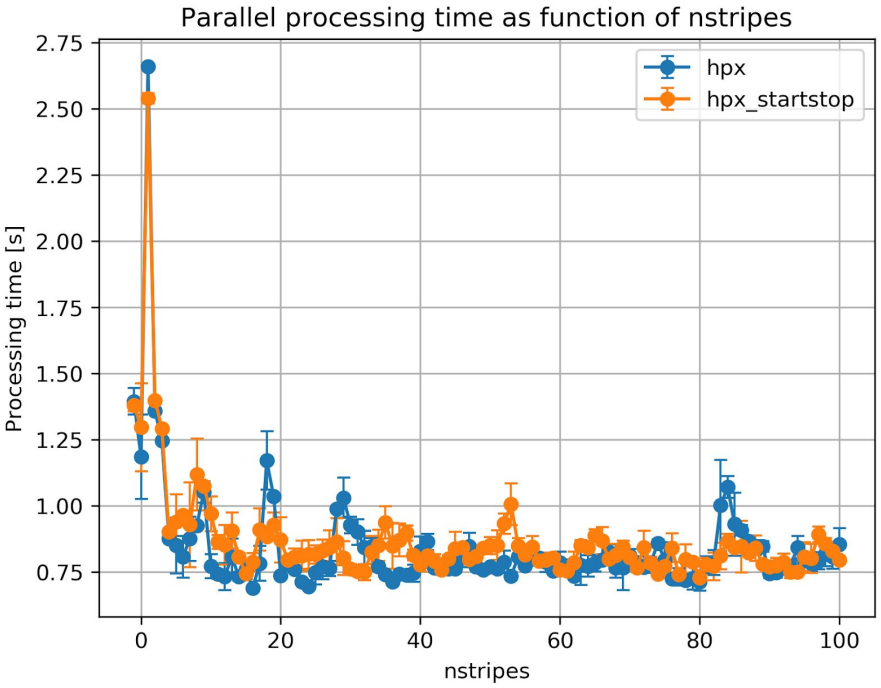
Nevertheless we are aware, that it creates a problem for the users who want to use the HPX backend without having to worry about runtime management. Therefore, a second version of the backend was developed which automatically starts the HPX runtime for each cv::parallel_for_() call. Currently the implementation is simple and does not perform any checks whether the runtime was started or not and blindly starts at entry and stops it on exit from cv::parallel_for_(). In the next month I plan to attempt developing a more advanced implementation which will perform checks on a static object in order to prevent starting the runtime multiple times. From now on I will refer to this version of backend as the *start-stop* backend.

It is worth pointing out that the start-stop version is not optimal. As mentioned before HPX is a general purpose framework for parallel and distributed computation with a DAG workflow. Therefore, starting and stopping it for each for-join style cv::parallel_for_() is introducing a considerable overhead as can be seen in the following benchmark I performed:



In the plot above it is visible, that the best performance is achieved by pthreads backend developed within OpenCV for the sole purpose of supporting the cv::parallel_for_() calls. What we can see in the graph above is the trade-off between generality of the solution (hpx) and tuning performance for a specific use-case (pthreads).

Apart from working on the backend itself I developed a set of example applications in order to both familiarize myself with HPX and OpenCV but also to test and benchmark different backend versions (see my repository). For example at first it was not clear to me what the *nstripes* parameter of the cv::parallel_for_() function is and how to handle it in the HPX backend, but after running a few benchmarks and analyzing results I chose the best out of different possible solutions. The *nstripes* parameter allows the OpenCV to give a hint to the parallel backend about the preferred partitioning of the tasks passed to cv::parallel_for_(). In the final implementation I take into account the *nstripes* partitioning but allow at maximum 4*num_threads of HPX tasks to execute all the *nstripes* partitions. For example if OpenCV uses *nstripes* equal to 100 and the HPX runtime is working on 8 threads it will still use 32 HPX-tasks and divide 100 chunks of work between them. It allows to avoid situations in which too high number of tasks is introducing too big overhead.



Finally, after a reliable version of the HPX backend for OpenCV was developed I used the OpenCV with freshly added backend for an example face-recognition application that works in real time. This application is the example in which usage of primary HPX backend is exemplified - as the user I have the full control of the HPX runtime and in particular over HPX thread pools. In this application I am using 2 thread pools: the default thread pool which receives 7 threads and the opencv thread pool which receives 1 thread. All the potentially blocking tasks are executed by opencv thread pool (showing each frame of the image, accessing the webcam in order to take each image of the video stream). The face-recognition algorithm from OpenCV library is run on the default thread pool and makes use of 7 threads assigned to this pool. Below is the gif presenting the application in use: