FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Implementation of a backend to ISPC using HPX

Bachelorarbeit

Julian Wolf

Friedrich-Alexander Universität Erlangen-Nürnberg
Chair of Computer Science 3 (Computer Architecture)

July 13, 2016

ABSTRACT

The following thesis first takes a closer look at parallelisation of programs in combination with hardware architecture like multiprocessing and vector instruction sets. Classifications and differences are explained, as well as some popular solutions. The two main software components, ISPC and HPX, will also be introduced and explained here. Afterwards the implementation of a HPX backend for the asynchronous task system interface of ISPC is explained and two different versions developed which later get compared using different test programs like a simple benchmark, a Mandelbrot set and a sobel image processing edge detection filter. Tests and benchmarks show that the HPX task system backend is an equal option to other solutions like pthreads or OpenMP.

CONTENTS

1 INTRODUCTION

Due to development of computer architecture new programming models and parallelisation play a more and more important role and have a big impact on computing performance. In the following thesis two major parallelisation categories, parallelisation through multiprocessing and vectorisation, will be brought together.

After covering the necessary background about parallelisation and categorisation, the components of this thesis are looked at more closely. The ISPC compiler offers a programming model for easy vectorisation, but also offers an open interface for task based (multiprocessing) parallelism. HPX is a library for multiprocessing and distributed computing. The goal is to implement and test a HPX based backend and task system for the ISPC compiler. In the course of the thesis two different versions are implemented, one using a HPX parallel algorithm and one using HPX asynchronous tasks.

Afterwards this task system is tested with a theoretical benchmark, a Mandelbrot set and an image processing edge detection filter and compared to already existing backends using OpenMP and pthreads. While one of the HPX implementations performs significantly slower, the other version performs equally to the existing backends and provides a good new option.

2 BACKGROUND

2.1 PARALLELISATION

In the development of computer architecture, increased processing unit performance was achieved by increasing the clock rate for a very long time. Due to physical boundaries there are limits for further increasing processor clock rate, therefore more speedup can only be achieved by more processing units and parallelisation of applications.

Another factor playing a role here is the so called Polack's rule. This rule states that the processing capacity grows proportionally to the square root of the increased complexity (number of transistors, area)[4]. This rule applied backwards means that with half the complexity of a processor it still has about 70% of the computing power. So while on the one hand neither increasing the clock rate nor increasing processor size and complexity are reasonable in a large scale and on the other hand with big progress in manufacturing and technology, development for modern processors went to multicore architecture with several smaller processing units and parallelised programs.

2.2 CLASSIFICATION OF PROGRAMS

With development of parallelisation several different approaches came up and therefore a classification of programs was promoted. The classification is based on data and instruction streams with a difference in a single or multiple streams. So there can be single instruction single data (SISD), single instruction multiple data (SIMD), multiple instruction single data (MISD) and multiple instruction multiple data (MIMD). Additionally a separation between the number of programs used for executing on multiple data can be made, single program multiple data (SPMD) and multiple program multiple data (MPMD)[5]

One approach is vectorising programs, loading multiple data and executing the same instruction(s) on them. This is a typical SIMD approach and very common within modern processors (e.g Intel SSE, AVX). Another scenario is running several instances of a single program in parallel, which is called SPMD (single program multiple data).

Vectorisation (SIMD) is often used through specific compiler instructions, which makes this type of parallelisation harder to use and not very portable. Single program multiple data approaches on the other hand usually come with their own programming concept and are more portable and flexible. This classification is also used throughout this thesis as SIMD and SPMD play a big role.

2.3 PROBLEM DESCRIPTION

Modern day computing is in need of parallelisation to further improve program performance and adapt to changing hardware technology. With multicore architecture becoming more and more important as well as rapidly growing number of processing units within systems, it is necessary to adjust the software architecture. For graphics processing on GPUs, where more processing units are available and used than in CPUs, new programming models are already available and widespread. CUDA and OpenCL are examples for this. For CPUs exist many concepts and ideas on how to achieve good parallelism as well, but this field still offers a lot of room for improvement.

On the one hand, parallelisation through multiprocessing gives a lot of performance and can be somewhat easily achieved through a variety of existing libraries. But the speedup of multiprocessing also depends a lot on which parts of a program can be parallelised and how big the serial part of a program is. Because the bigger the serial part of a program, and that might not always be changeable, the less influence parallelisation of the rest of the program has on the overall speedup.

On the other hand, if big parts of the program can be parallelised, the resulting speedup will be very good. Gene Amdahl presented a law, named after him (Amdahl's law), which discusses the relationship between serial and parallel part of a program and resulting possible and theoretical speedup. The law can be displayed as a formula

$$s = \frac{1}{r_s + \frac{r_p}{n}}$$

where s is the theoretical speedup, r_s is the sequential part, r_p the parallel part and n the speedup which can be reached through parallelisation of the parallel part. In sum r_s and r_p must equal 1[3]. The speedup is relevant for all types of parallelisation, be it vectorisation with SIMD or multiprocessor parallelisation, and therefore important as both types will be discussed later.

SIMD parallelism and vectorisation, as already quickly introduced earlier, are still an area that often isn't used to it's full potential. While the programmer has a lot of support for thread programming through libraries, SIMD programming (vectorisation) often is handwritten via assembler or compiler provided SIMD intrinsics. The major drawbacks of this approach are clearly that code is not portable without adapting, as different hardware has different SIMD capabilities and, related, the code is harder to maintain due more difficult readability, required expertise and more necessary changes. And while the major C/C++ compilers do have functionality implemented for auto-vectorisation, e.g. for loops, it's success depends a lot on factors like loop or data layout[9]. Therefore a more abstract approach, that doesn't rely on the compiler and auto-vectorisation and gives more freedom and independence of hardware, is needed in the area of SIMD programming. One approach, the Intel® SPMD Program Compiler will be discussed in section 2.5.

2.4 HPX

HPX, which defines itself as a "general purpose C++ runtime system for parallel and distributed applications of any scale"[8], tries to provide a new model that solves many problems in current and future high performance computing. HPX is committed to provide a solution that does focus on *scalability*, *programmability*, *performance portability*, *resilience* and *energy efficiency*[8]. HPX is a C++ library (within the C++11 standard) and provides the user with a model for decentralised, task based and scalable parallelism.

Any approach for a scalable solution in this area will have to face several obstacles on this way, the so called *SLOW* factors: *Starvation*, *latency*, *overheads* and *waiting for contention resolution*[8]. The next paragraph intends to give a quick overview over the design principles behind HPX to tackle the *SLOW* factors[8].

- Instead of focusing on latency avoidance, which can never be totally achieved, HPX focuses on using latencies for other work, means latency hiding.
- Use of lightweight thread and fine grained parallelism instead of heavyweight threads. This results in shorter context switches and supports the goal of hiding latencies which helps with better system utilisation.
- Rediscovering constrained based synchronisation as a replacement for global barriers as they often can be found in modern systems to keep the parallel part as big as possible and reduce the sequential and slower part to a bare minimum.
- In big computing clusters data locality is very important and HPX targets this area with a global and uniform address space and more dynamic data distribution among the nodes.
- Moving as little as possible between nodes, means to move work to the data instead of (big amounts of) data to the computing nodes.

- HPX also focuses on message driven computation, which works with asynchronous messages and work queues instead of using message passing, which requires a lot of synchronisation for passing messages.

2.5 INTEL®SPMD PROGRAM COMPILER (ISPC)

The Intel® SPMD Program Compiler (ISPC) is an open-source (BSD-license) compiler focusing on single program multiple data (SPMD) programming. ISPC compiles a C-based language which supports SPMD programming concepts to use the SIMD units on the CPUs. Besides of SIMD level vectorisation, ISPC also supports task parallelism to make use of multicore architecture. ISPC uses the LLVM compiler infrastructure for code generation and optimisation.

An ISPC design goal was to create a compiler for modern SIMD CPUs which does not only rely on auto-vectorisation for a sequential language during compile time, but also to provide parallel semantics and a programming model for complex control flows.

Executing multiple instances of a single program in parallel, SPMD, which is similar to concepts used by GPU shaders or CUDA, are now used by ISPC. Those concurrent program instances are bundled in a group, called *gang*.The number of instances within a gang is determined at compile time for the hardware but not bigger than twice the size of the hardware's SIMD width. The gang is also running in the same context and thread as the application code and does not create more threads or do context switches.

ISPC also tries to create performing code from loops or branched control flows by allowing different members of a gang (program instances) to execute different paths. It also offers a lot of options to the programmer to control behaviour of those parallel loops, that might, depending on the situation and code, be used to further increase performance. Therefore ISPC offers, besides a parallel for-loop, additional loops like *foreach, foreach_tiled, foreach_active, foreach_unique*.

ISPC's SPMD programing model also includes the concept of asynchronous tasks that can be executed asynchronously on different threads, which can be distributed among processing units. This equates to the fork-join concept, as the tasks act independently and get joined explicitly or implicitly when leaving the function which launched the tasks. Tasks can be launched in up to three dimensions and each task/thread can be identified via an unique ID. With this concept it is possible to launch a number of tasks, which then, depending on problem size, task count and own task identification, calculate a part of the big problem. This is very similar to the model used in CUDA for NVIDIA graphic card programming, which also provides three dimensions of thread hierarchy and something similar to tasks, called kernels, that get executed in parallel[1].

While ISPC doesn't provide a task system or thread-pool itself, it offers an easy to use interface for launching and syncing tasks which can be used with any backend[2][10].

The programming model of ISPC is also a big difference to other SIMD libraries, like *Vc*. While Vc also focuses on vectorising instructions and on abstracting assembler code and compiler intrinsics, it only provides a layer to C/C++ code without adding new concepts. It also doesn't touch multiprocessor support, as this is already available via other libraries like OpenMP. So Vc only focuses on making SIMD programming easy, accessible and portable without changing major concepts for the programmer. A very good example for the layer of abstraction provided by Vc is the introduction of data types like *float_v* (a float vector) which replace the intrinsics that depend on the size of registers. Both approaches certainly have their advantages and disadvantages and therefore use cases for both can be found[9].

The details and requirements on how a backend for the ISPC task system will be discussed later in this paper. Goal of this thesis is to create such a backend for ISPC using the HPX framework.

2.6 COMBINING ISPC AND HPX

As discussed in the previous part, the ISPC language offers support for task based parallelism and HPX is a programming model with heavy focus on task based parallelism as well. Therefore the idea of using the advantages of the ISPC language and compiler to create tasks which can then be distributed and scheduled through the HPX system as backend. Gladly ISPC offers a programming interface which makes it easy to implement or attach task systems as a backend. The next chapter will have a closer look at the interface and implementation.

3 IMPLEMENTATION

3.1 ISPC TASK PARALLELISM RUNTIME REQUIREMENTS

ISPC offers task parallelism through *launch* and *sync* statements without specifying the runtime or task system behind it. To make use of the task parallelism three functions must be provided and linked as *C* functions. This gives a big degree of freedom for different backend systems and own task management implementations. In this thesis HPX was included as a backend system which offers an easy to use and very flexible environment and support for distributed computing. The three functions that need to be implemented and linked for ISPC's *launch* and *sync* statements are bound to the following signature:

```
void *ISPCAlloc(void **handlePtr, int64_t size, int32_t
    alignment);
void ISPCLaunch(void **handlePtr, void *f, void *data,
    int count0, int count1, int count2);
void ISPCSync(void *handle);
```

Listing 3.1: ISPC task system runtime requirements

The first argument to all three functions is a unique handle which helps the task system distinguish tasks from different calling functions. The *ISPCAlloc* function only serves the purpose of allocating memory of a specified size. The *ISPCLaunch* function is used to launch tasks. For this it is provided with a function pointer *f*, arguments to that function *data* as well as the number of tasks to be launched for up to three possible dimensions *count0, count1, count2*. The function signature for the task looks as follows:

```
void (*TaskFuncPtr)(void *data, int threadIndex, int
    threadCount, int taskIndex, int taskCount, int
    taskIndex0, int taskIndex1, int taskIndex2, int
    taskCount0, int taskCount1, int taskCount2);
```

Listing 3.2: ISPC task function signature

The arguments here are for providing a single task with information about the numbers of threads and tasks and its own position.

The function *ISPCSync* as third and last function to be implemented only expects a handle as argument and only waits for all tasks to finish before exiting.

3.2 ISPC PROVIDED TASK SYSTEM

ISPC already has a implementation for those functions and also provides a good structured and easy to extend task system. Before going deeper into how it got extended, first a look on the already existing structure[1]. Part of it is a structure called *TaskInfo*, which saves every relevant information for a task, among them the function *f* to be called and the arguments. There is also a class *TaskGroupBase* which provides basic functions like the constructors, destructors and allocation of memory for a group of tasks. This task then can easily be extended by any type of backend through a class called *TaskGroup* which provides backend specific *Launch* and *Sync* functions.

So the implemented ISPC functions only need to initialise the task system if necessary, assign *TaskInfo* information and call the corresponding *TaskGroup::Launch* and *TaskGroup::Sync* statements. There are already implementations for several different backends like OpenMP or pthreads and those now got extended with one for HPX.

3.3 HPX EXTENSION

During the development two different ideas came up on how to create and distribute asynchronous tasks with HPX. Both shall be explained here, including basic code, and be compared as well as analysed for performance later.

The first idea relies on a basic loop and the *hpx::async* function for creating the asynchronous jobs. This has the advantage of being a straightforward and easy solution at the disadvantage of having a future object per task creation and synchronising all tasks via all the futures later. The implementation here uses a vector of futures for storing the asynchronous tasks return values.

```
futures.push_back(hpx::async(ti->func, ti->data,
    threadIndex, threadCount, ti->taskIndex,
    ti->taskCount(), ti->taskIndex0(),
    ti->taskIndex1(), ti->taskIndex2(),
    ti->taskCount0(), ti->taskCount1(),
    ti->taskCount2()));
```

Listing 3.3: Launching asynchronous tasks

1 https://github.com/ispc/ispc/blob/master/examples/tasksys.cpp

For the synchronisation an extended operation for futures which is offered by the HPX library is used and allows easy access to several ways how to handle futures. Here the *hpx::wait_all* function is used to synchronise the tasks.

```
hpx::wait_all(futures);
futures.clear();
```

Listing 3.4: Synchronising tasks

The second approach uses a HPX function *for_each* which basically is a parallel loop performing some action over a number of elements. This has the advantage that the function returns exactly one future object for synchronisation and also offers a wider range of flexibility due to selectable execution policies. Those execution policies can be anything from sequential to parallel and asynchronous and can even be extended for other or special purposes. The following piece of code shows the part of launching tasks using the parallel loop and storing the future object.

```
future = for_each(
    par(task),
    boost::begin(range), boost::end(range),
    [=](int i){
        TaskInfo *ti = GetTaskInfo(baseIndex + i);
        int threadIndex = i;
        int threadCount = count;
        ti->func(ti->data, threadIndex, threadCount,
             ti->taskIndex, ti->taskCount(),
             ti->taskIndex0(), ti->taskIndex1(),
             ti->taskIndex2(), ti->taskCount0(),
             ti->taskCount1(), ti->taskCount2());
    });
}
```

Listing 3.5: Launching asynchronous tasks

In this case the synchronisation is very easy as there is only one future object to wait for.

```
future.wait();
```

Listing 3.6: Synchronising tasks

The index based parallel for-loop used here is a feature introduced by HPX to overcome some weaknesses traditional parallel loops have, especially in regards of indexes. Therefore HPX provides a collection of parallel algorithms, like *for_each*, and gives the programmer a very powerful and generic tool. With the use of several execution policies the parallel loop can also be customised depending on the situation and use case. HPX offers both sequential and parallel execution policies, but also a task based asynchronous one, *par(task)*, which is used above to launch the task function asynchronously[7].

The only requirement for using the HPX task system for ISPC is, that the main program from which the task system is used has started and initialised the HPX runtime. In all following examples the code of the main program got extended by a conditional include statement which is activated during compile time and automatically uses the main-function as entry point for HPX, but launching HPX with custom functions is possible as well.

```
#ifdef ISPC_USE_HPX
        #include <hpx/hpx_main.hpp>
#endif
```

Listing 3.7: Initialise HPX environment with start of main-function

4 RESULTS

To compare both variants with each other as well as with other, already implemented, backends for the ISPC task system several tests were run to generate comparable results for both benchmarking in relation to size of tasks as well as number of tasks and real-life programs.

The duration of the ISPC function from call to return is measured in milliseconds. The implementation for the timing function used is implemented and distributed by Intel with ISPC[1].

As the main part of this thesis is about CPU parallelisation and the tests are also CPU-bound, all the tests were run on different platforms to provide comparability. Differences are not only by number of cores or hardware threads, but also by available vector instructions like AVX/SSE. Therefore the results can hardly be compared to each other in absolute numbers, but only within one platform. The comparison between platforms can only show a general trend. As the already existing backend implementations used here, using OpenMP and pthreads, determine the number of created operating system threads themselves but HPX has this configurable through a program parameter *–hpx:threads*, the number of created threads used for the following tests always is equal to the number of available processing units by specifying *all*. So for example on the Intel i7 Skylake processor with four cores and two threads per core, eight threads were used and on the AMD Opteron with two sockets and two cores per socket four threads were used.

Three different hardware platforms were used for the following tests and measurements. One, an Intel i7 Skylake processor with four cores and two threads per core, an Intel Pentium N3540 notebook processor, and an AMD Opteron. All details and specifications can be found in table 1.

1 https://github.com/ispc/ispc/blob/master/examples/timing.h

Model name	Intel i7-6700K	Intel N3540	AMD Opteron 2216 HE
Architecture	x86_64	x86_64	x86_64
CPU(s)	8	4	4
Thread(s) per core	2	1	1
Core(s) per socket	4	4	2
Socket(s)	1	1	2
CPU max MHz	4200.0000	2665.6001	2399.998
L1d cache	32K	24K	64K
L1i cache	32K	32K	64K
L2 cache	256K	1024K	1024K
L3 cache	8192K		

Table 1: CPU specifications

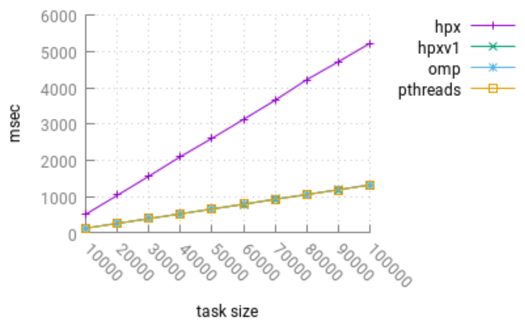
4.1 TESTS AND PERFORMANCE MEASUREMENT

4.1.1 *Benchmark*

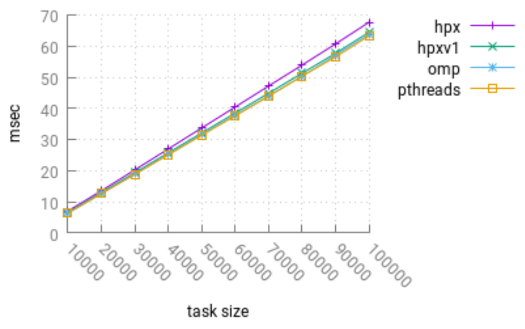
The first series of tests is a very simple benchmark that launches *n* tasks where each tasks calculates each reciprocal for *m* (pseudo-) random numbers. As both variables, size of a task and amount of tasks, are interesting the tests were split into three parts, with constant task size and increased task count, constant task count and increased task size and last with both increased task size and count. For the first two parts linear graphs are expected due to linear increase of parameters, in the last case polynomial or exponential growth is expected. In the following part the results shall be displayed, discussed and analysed.

The first figure 1, which displays the graphs for a constant task count of 1000 and a variable task size up to 100000, shows that the backend implementation starting tasks via *hpx::async* and storing all returned futures in a vector and synchronising with *hpx::wait_all*, which shall be referenced as version 1 from here on, is on both a Intel N3540 with four cores and a Intel i7 with four cores and hyperthreading about on the same level as the already available implementations for OpenMP and pthreads. A significant difference can be made out between the two different HPX implementations, especially on the slower processor where the second version is several times slower than the first one. On the significantly more powerful i7 processor the difference is a lot smaller, but version 2 is still clearly slower.

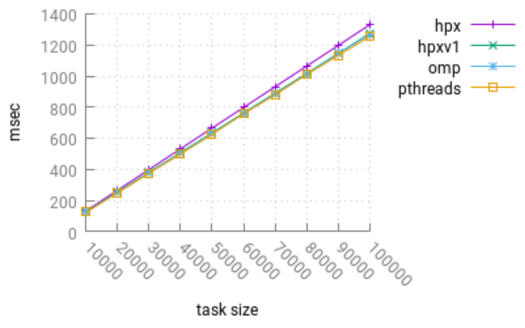
Figure 2 shows the test results for a constant task size and a variable amount of tasks, up to 100000. The first observation which can be made here is, that on the slower N3540 processor the second version of HPX is by far slower than all other implementations, while on the faster i7 processor the first implementation is slower. Another observation which can be made here is that both HPX implementations



(a) Intel N3540



(b) Intel i7



(c) AMD Opteron

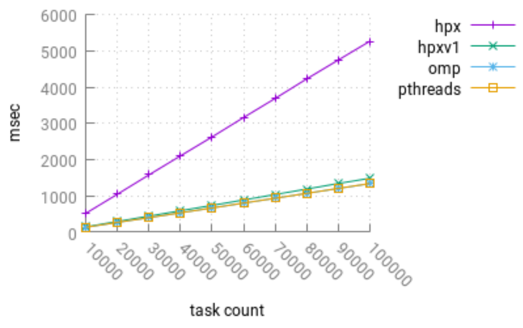
Figure 1: Fixed task count (1000), task size variable

are, on all tested processors, slower than the OpenMP and pthread implementations.

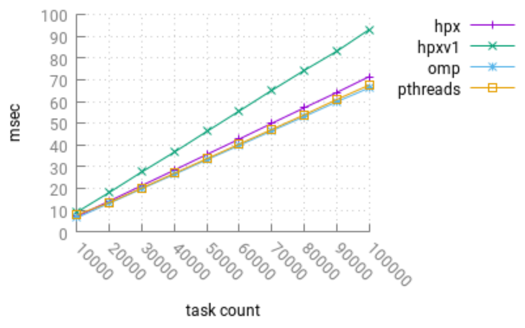
The last test case in this benchmark, which can be found in figure 3, shows the results for both the values for task size and task count being linearly increased up to 50000. As expected, all four tested implementations show polynomial growth. On the i7 and Opteron processor, the second HPX implementation is slower than the other three variants. On the N3540 the second HPX implementation is far off from all other options and significantly slower. The other three backends gave very similar results here.

Due to some hints that the second HPX version with the parallel for-each algorithm being slower with smaller and more fine grained tasks a set of this test was also run to further inspect this behaviour. For this, the program was run several times with different amount of tasks and with bigger task sizes (up to 300000 calculations on the i7 processor). And, as already indicated and therefore expected, the behaviour could be partially confirmed. On the Intel i7 processor all task system backends behaved very similar up to around 800 tasks, then the parallel algorithm HPX backend, the second version, started to become significantly slower than the three other versions.

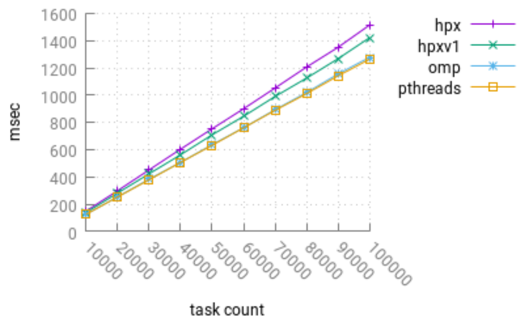
On the weak Intel N3540 processor on the other hand this behaviour could not be reproduced, the second HPX version behaved worse even in situations with small amounts of tasks. The task sizes



(a) Intel N3540

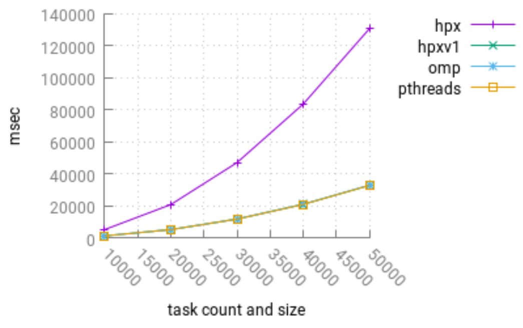


(b) Intel i7

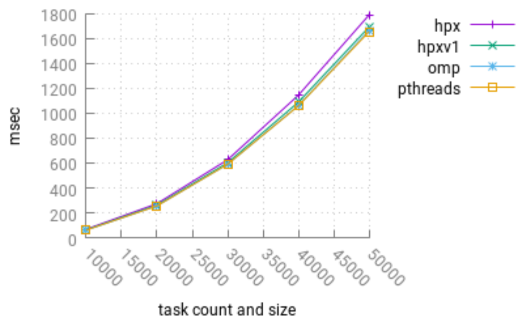


(c) AMD Opteron

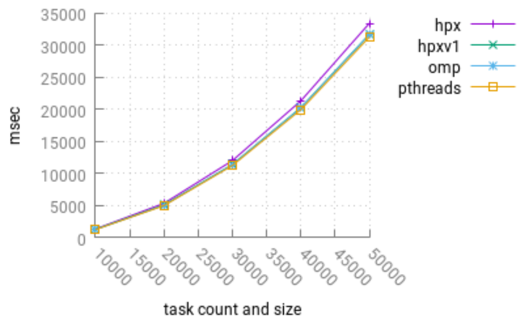
Figure 2: Fixed task size (1000), task count variable



(a) Intel N3540

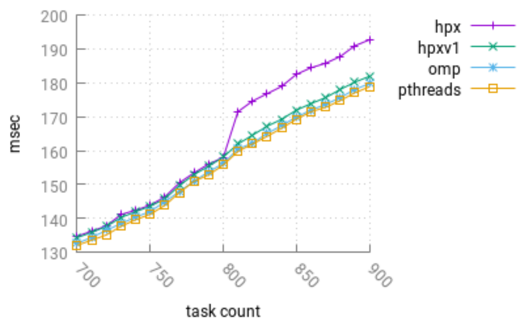


(b) Intel i7

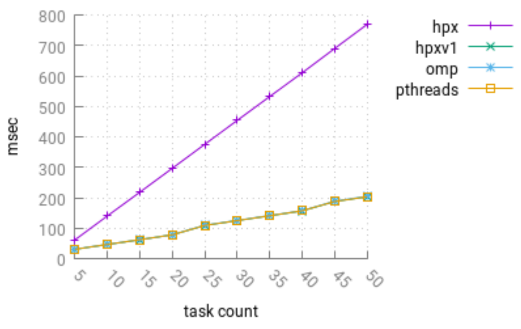


(c) AMD Opteron

Figure 3: Both task size and count variable



(a) Intel i7



(b) Intel N3540

Figure 4: Comparison with small task counts and big tasks

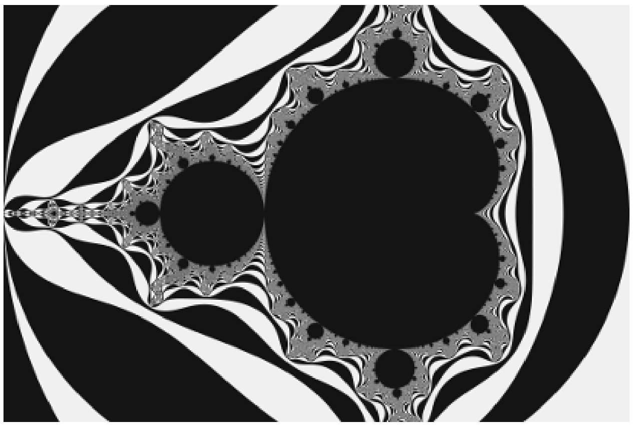


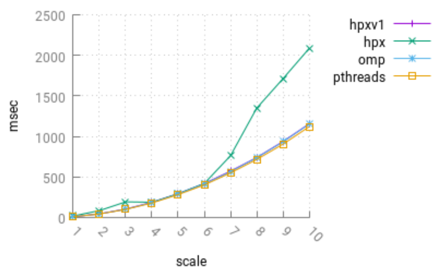
Figure 5: Mandelbrot

used here were significantly smaller and adjusted to the computing capability of the processor. Both observations can be found in figure 4. The graphs are scaled accordingly, the i7 processor graph is scaled to the point where the overhead of too fine task granularity can be observed, the N3540 graph is scaled to show that the overhead is too big from the beginning.

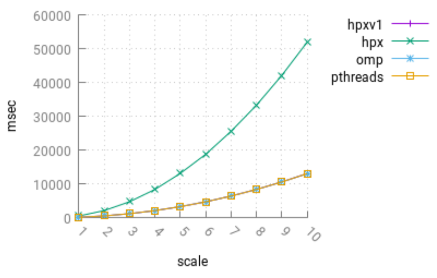
4.1.2 *Mandelbrot set*

The next series of tests were run with an already available implementation of the Mandelbrot set using ISPC and tasks and which is also shipped with ISPC as an example for this. The Mandelbrot set implementation allows to set the scale, and with that the size of the computed image and also the amount of work and tasks to be set as a parameter. The default size is 1536x1024 which is then scaled with a parameter (seen on x-axis in figure 6) and rounded. The task count is defined by a quarter of the height. A resulting Mandelbrot image can be seen in figure 5.

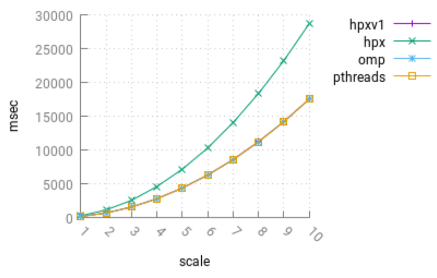
All results of the Mandelbrot set, as can be seen in figure 6, produce polynomial curves (the HPX version 2 on the i7 having an out-



(a) Mandelbrot on Intel i7



(b) Mandelbrot on Intel N3540



(c) Mandelbrot on AMD Opteron

Figure 6: Mandelbrot set using ISPC tasks

lier part). The second HPX version is here under all circumstances significantly slower than the other three tested variants and shows way faster growth. The first HPX version, the OpenMP version and the pthread version on the other side show very similar results again and lay almost on top of each other.

Very interesting to see here is a jump in the graph of the second version of the HPX implementation on the i7 processor. Until a scale factor of six, the performance is similar to the other three implementations, after that it is significantly slower and the graph grows faster. This goes hand in hand with the final comparison of the previous benchmarking test which also shows a jump when the amount of tasks used exceeds a certain threshold.

4.1.3 *Sobel filter*

As a last test using ISPC tasks an image processing sobel filter for edge detection was implemented. The algorithm calculates the new value of a single pixel by applying two kernels, one for x- and one for y-direction, to the surrounding pixels.

The ISPC function creates n tasks, with n being the number of rows the input image has. Each task then uses the *taskIndex* variable to identify itself and find the row it is responsible for. Afterwards the task iterates through the row using the ISPC *for_each* loop and applies the sobel kernel on every single element in the row. The sobel filter main program uses the OpenCV graphic processing library for input

(a) beach, 4608x3456, 15.9MP

(b) parrot, 4608x3456, 15.9MP

(c) hubble, 15852x12392, 196.4MP

(d) earth, 8000x8000, 64MP

Figure 7: Sobel filter input images

and output of images, but not for actually processing the image, this part is implemented in ISPC code.

For this test four different images with a significant resolution were used, two smaller ones, parrot and beach, with a resolution of 15.9MP and two bigger ones, earth[2] with 64MP and hubble[3] with 196.4MP. The input images and dimensions can be seen in figure 7. As the sobel filter implementation with ISPC used here creates one task per row, the amount of tasks depends one the image dimensions. As the focus of this thesis lies in the implementation of the HPX backend and comparing it to other, previous implementations, it was not the goal to create an ideal implementation of the sobel filter. Better solutions, especially considering and finding the perfect amount and size of tasks, can certainly be found. The produced output images are shown in figure 8.

As can be observed in figure 9, on the Intel N3540 processor the second HPX implementation was significantly slower by a factor greater than three with all four images. All other three backend implementations performed very similar and did not show any significant differences in performance. On the Intel i7 processor, which can be seen in figure 10, the second HPX implementation was slower with all four test cases as well, but the difference is a lot smaller and less significant. The other three implementations show small and varying differences, but considering how small the differences are they can be

2 Credit: NASA, http://www.jpl.nasa.gov/spaceimages/details.php?id=PIA18033
3 Credit: NASA, http://hubblesite.org/newscenter/archive/releases/2006/10/image/a/

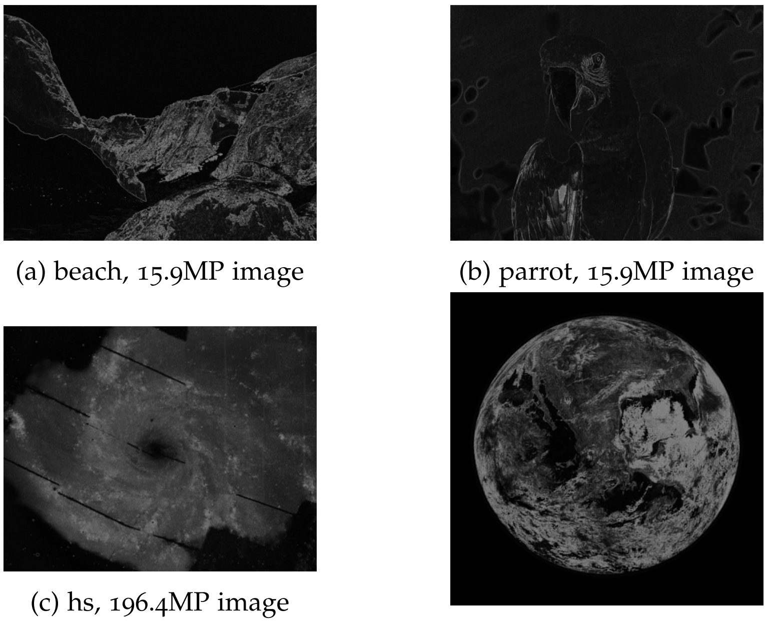

(a) beach, 15.9MP image
(b) parrot, 15.9MP image
(c) hs, 196.4MP image
(d) earth, 64MP image

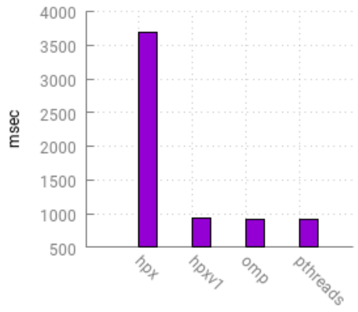
Figure 8: Sobel filter output images

seen as equal and it's not possible to make conclusions based on the available data.

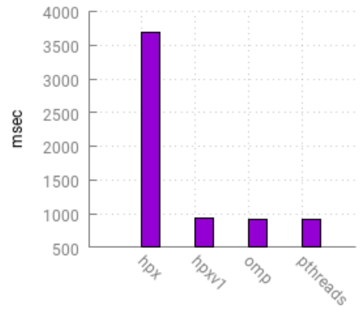
4.2 INTERPRETATION

The first result from the previous tests is, that the second HPX version is in almost any test case slower than the first version or the other two already existing versions using OpenMP and pthreads. The interpretation and reasons for this are more complex though, as a variety of factors play a role here. With a first factor definitely being the processing unit and it's capabilities. On the rather slow Intel N3540 processor the first HPX implementation was significantly slower, often by a multiple, under almost all circumstances. The other three tested implementations were very close to each other, only the implementation using the HPX parallel for-each algorithm was a real outlier here. As the graphs are way closer to each other on the other tested CPUs, it indicates that the overhead has bigger impact on weaker processors.

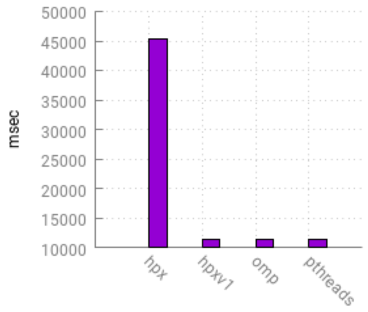
A second observation is the role of the number of tasks. This could be observed very good on the Intel i7 processor running the Mandelbrot set and running the benchmark program with very big tasks. On the Mandelbrot set graph it can be seen that until a scale of six, the second HPX solution performs similar to the other implementations, but then a big jump significantly drops the performance. In the benchmark this same jump can be seen when keeping the task size constant and big, and slowly increasing the amount of tasks. The point the jump happens here is around 800 tasks. Why this can only



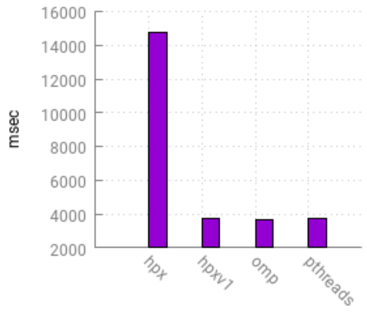
(a) Intel N3540, 15.9MP image



(b) Intel N3540, 15.9MP image

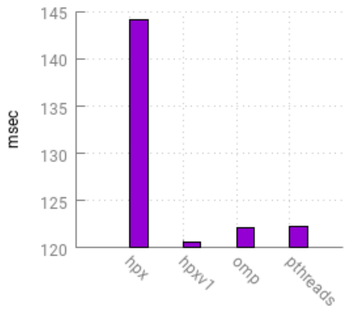


(c) Intel N3540, 196.4MP image

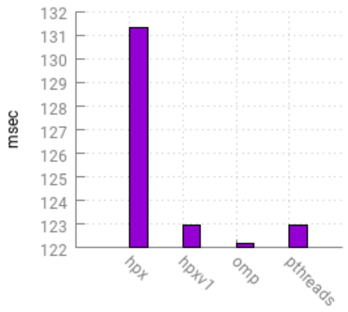


(d) Intel N3540, 64MP image

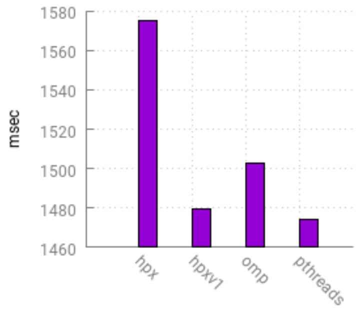
Figure 9: Sobel filter on Intel N3540



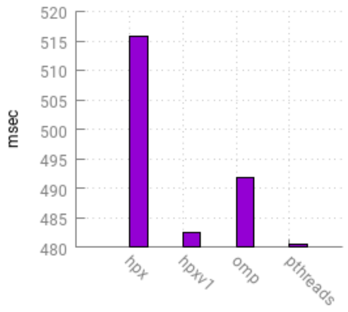
(a) Intel i7, 15.9MP image



(b) Intel i7, 15.9MP image



(c) Intel i7, 196.4MP image



(d) Intel i7, 64MP image

Figure 10: Sobel filter on Intel i7

been seen on the i7 processor and not on the others can, without further investigation, only be speculated. But this new Skylake processor with a 8MB L3 cache might be able to cover up some overhead latencies up to a certain point. Higher task count also has impact on the first HPX version, as can be seen in figure 2. Both HPX versions perform worse than the OpenMP and pthreads solution on all tested CPUs in this case, on the i7 processor the first HPX version even performs worse than the second version. This is the only test where this can be seen, so it seems like the factor CPU is playing a big role here, too. The relationship between task granularity and performance in HPX was already topic of a work by Grubel, Kaiser, Cook and Serio in 2015, with the result being that both very small tasks and very big tasks having a significant negative impact on execution time due to big overhead in the first case and bad load balancing and scheduling in the case of big tasks[6].

While neither HPX backend implementation was able to outperform the already existing backends, the first HPX implementation did perform equally in almost all cases. The only scenarios where a significant difference could be noted was with very high task counts and fine granularity, as can be seen in figure 2. In this case the HPX overhead decreased performance compared to OpenMP and pthreads. But considering how easy it is to use ISPC functions from within a program that might already run HPX and that the performance is equal in many cases, using a HPX backend for ISPC tasks is a good option even though no performance gains could be shown.

5 CONCLUSION

As discussed in the first part of this thesis, parallelisation is a topic that involves both writing programs for multiprocessor architectures and for using vector instructions available in modern processing units for SIMD parallelisation. While a major focus is on multiprocessor parallelisation and a lot of libraries and solutions exist for this, less attention is payed on SIMD parallelisation which leads to fewer research or solutions. SIMD parallelisation is often only reached through compiler intrinsics, which is hard to maintain and hardware dependent and therefore not portable.

While Vc is a library that offers an additional abstraction layer for easier and portable vectorisation, ISPC offers a CUDA-like programming concept with multiple program instances (*gangs*) and also a concept for task based multiprocessor parallelism. ISPC doesn't specify the backend design and offers a function interface to attach custom backends.

In this thesis a backend using the HPX library for parallel and distributed applications was created. In the process two different approaches were implemented, one using asynchronous tasks and one using a HPX parallel algorithm. Both variants were compared with already existing backend implementations provided by ISPC using OpenMP and pthreads. To test and compare the backends three different test programs using ISPC tasks were used.

First, a benchmark with both variable task size and task count, second a Mandelbrot set program provided by ISPC, and third a sobel image processing filter for edge detection. The first HPX backend implementation performed better than the second one in most cases and similar to the OpenMP and pthreads implementations. For the overall performance several factors play important roles. Big differences between used CPUs could be observed, but also the relationship between task size and amount of tasks needs to be considered for applications. Fine granularity led to bigger overhead in the HPX backend and therefore worse performance.

In general, the HPX version 1 performed very similar to the already existing implementations and can be considered equal if a balance task granularity can be assured. As it is easy to include ISPC code in C/C++ code using HPX and additionally take advantage of the easy to use ISPC task system, which can also be based on HPX, it can be considered a good option to other popular parallelisation solutions. The first task system version of HPX was submitted and accepted to the ISPC Github repository[1].

1 https://github.com/ispc/ispc

6

BIBLIOGRAPHY

[1] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide. Accessed: 2016-06-08.

[2] Intel® SPMD Program Compiler User's Guide. http://ispc.github.io/ispc.html. Accessed: 2016-04-14.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[4] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[5] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[6] P. Grubel, H. Kaiser, J. Cook, and A. Serio. The performance implication of task size for applications on the hpx runtime system. In *2015 IEEE International Conference on Cluster Computing*, pages 682–689, Sept 2015.

[7] H. Kaiser. HPX and C++ Parallel Algorithms. http://stellar-group.org/2015/06/hpx-and-cpp-parallel-algorithms/, 2015. Accessed: 2016-06-09.

[8] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX – A Task Based Programming Model in a Global Address Space. In *PGAS*, 2014.

[9] M. Kretz and V. Lindenstruth. Vc: A c++ library for explicit vectorization. *Software: Practice and Experience*, 42(11):1409–1430, 2012.

[10] M. Pharr and W. R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *InPar*, 2012.