Task-based Analysis of OpenMP and HPX

Tobias Schiffmann
Universität Stuttgart
Stuttgart, Germany
st169736@stud.uni-stuttgart.de

***Abstract*—Task-based programming is used to parallelize irregular structures like recursive function calls. A prominent examples for run time systems allowing to utilize tasks are OpenMP and HPX. OpenMP is the de facto standard for shared memory parallel programming and in contrast to this HPX is a rather new run time system designed for task-based programming.**

These two systems have not yet been compared concerning tasking performance. To do so the Fibonacci, Merge Sort and a generic algorithm are implemented using both run time systems.

The performances of the executions are compared and some optimization ideas are evaluated.

It comes clear that both run time systems have structures they perform better with compared to the other system. HPX has problems managing high amounts of small tasks in hierarchical structure, however it achieves better performance with bigger tasks and non hierarchical structures.

***Index Terms*—High Performance Computing, HPX, OpenMP, Task-based Parallelism**

I. Introduction

Task-based programming increased in popularity in recent years. It allows to parallelize irregular and dynamic structures like while loops, dynamic linked list traversals and recursive function calls. [1] OpenMP and HPX are both parallel run time system which increased focus on task-based parallelism. [2] OpenMP, as de facto standard for shared-memory parallel programming did ~~not~~ increase its directive pool concerning tasks in the latest version, 5.0. [3] These two run time systems have not yet been compared in performance of task-based parallel execution. This paper tries till fill this gap.

The main focus is put on the task schedulers of both systems. Task schedulers play a crucial role as they have to manage load balancing and data locality. Having the same computation workload the differences in performance will emerge from these components. [4]

Section II will introduce the tasking principles and shows how they are used in both run time systems. It furthermore explains a new approach about using OpenMP tasks in HPX. Afterwards section III presents the used benchmark algorithms to compare OpenMP and HPX. Then a short overview of the experiment environment is granted, followed by the actual experiments in section VI. The two run time systems are compared to each other and a sequential implementation as reference. Further optimization options are tested and evaluated before a conclusion is drawn at the end.

II. Related Work

A. Tasking

Having the task construct as mean to parallelize a program allows to utilize irregular parallelism. This type of parallelism utilizes irregular data structures, e.g. linked lists rather than arrays. Furthermore communication phases and the needed amount of communication cannot be identified in advance. This is also the case for the amount of computation. [5] Parallel execution of while loops or recursive function calls perform irregular parallelism for instance. Tasking explicitly specifies independent units of work which can be executed in parallel. This enables a more dynamic way of parallelization ~~which~~ ~~can~~ handle irregular parallelism. However, a disadvantage of this kind of parallelism is that it increases the complexity of a program and poor implementations can lead to ~~an~~ increased overhead. [1] [6]

The used runtime system for parallel execution plays a crucial role when it comes to the application's performance using tasking. For example, as the tasking construct creates several tasks which will be executed by threads, the performance of an application relies on the thread scheduler of the runtime system. The authors of [6] discuss different aspects of a runtime system using the task construct. The efficiency of such depends on the data structures which store unfinished tasks, manage task switching and regulate task creation. Additionally, the data structures ~~to~~ organize task synchronization and manage the memory footprint of a task are also important. The ideal way of work ~~of the~~ thread scheduler, for example, is to maximize concurrency, load balancing and data locality. This can be achieved by different ways to manage queue storing all tasks ready to be executed. Furthermore, the scheduler can be *depth-first* or *breadth-first*. Breadth-first means that child tasks created by the parent task and put into the queue to be scheduled at any time. Whereas breadth-first schedulers switch to child tasks directly after their creation and execute them. This leads to a smaller number of tasks in the queue and less concurrency opportunities. However, the data reuse may increase depending on the running application.

The following subsections II-B and II-C introduce two runtime systems utilizing tasks. These will be compared in this work.

B. Tasks in OpenMP

OpenMP includes the tasking model since version 3.0. Various directives for that purpose are being included since

to specify and manage tasks. However, OpenMP still differs between explicit and implicit tasks. Implicit tasks are created by parallel regions as side effect. The programmer does not need to specify or know about them. Explicit tasks on the other hand are defined by the programmer using the task directives. The simplest directive to define an explicit task is `omp task` and the enclosed code area is the task region. These are executed by any task in the current team whenever they are ready. [1] [6]

When a thread encounters a task directive the current data environment is captured. This environment and the block of code in the task region form the generated task. The variable scopes of a task region can be defined like in parallel regions. By adding the clauses `shared`, `private` or `firstprivate` to the task directive the declared variables are either shared among tasks or not. [7]

By default a thread is tied to a task as it begins to execute it. This means that this task is only allowed to be executed by one thread. However this thread can still execute other tasks in case it reaches task scheduling points and switches the execution to another task. The suspended task is put into the queue and has to wait until its tied thread is ready to continue its execution. This restriction can be avoided by defining the task as `untied`. By doing so, load balancing is increased, but data locality is decreased. [1] [6]

Tasking results in a more dynamic execution and might be unpredictable without explicit scheduling. For example the runtime system has either a depth- or breadth-first scheduler. Meaning that it either switches to the child task after creation or finishes the parent task first. The used OpenMP runtime system is responsible to decide. Explicit scheduling can be used to avoid this uncertainty. For instance, adding an `if`-clause to the task directive and evaluating it to false makes the encountering thread to suspend its current work and execute the child task immediately. Furthermore, explicit task scheduling is possible by using the `taskwait` and `barrier` directives. `taskwait` suspends the encountering task region and forces it to wait for all child tasks to complete. Additionally `taskgroup` defines a task region which will be executed and at the end the current task has to wait for all its child and their descendants. [4] [8]

C. HPX

1) HPX: HPX aims to resolve problems in scalabilitiy, resource management and even further ones which may raise and increase by moving from Peta- to Exascale systems. It focuses on parallel and distributed development independent of the system's scale. This is done by providing a general purpose C++ runtime system with an innovative design. It is described by the authors of [9] as a mixture of lightweight synchronization, global system-wide address space and fine grain parallelism. Message driven computation can be achieved with a small amount of effort. A remote execution has the same semantics as a local one and can thereby be done implicitly. Furthermore, it offers the programmer explicit support for accelerators such as general purpose graphical processing units. [9]

A new model of parallel execution is introduced to enable all of this features — called *Parallex*. HPX is the runtime system supporting this new model. It consists of components which can be seen in figure 1. Additionally, it shows the current architecture of HPX and how the following ParalleX components are included. [10]

Parcel Subsystem encapsulates method calls into parcels. These contain the global address of the object whose method is called and the necessary arguments. Parcels are communicated via localities which are placed on each node and can be seen as parcel interfaces. Localities thereby represent single nodes in the execution network. The parcel subsystem is especially useful when it comes to remote method calls as it eases the work transfer between localities.

Active Global Address Space (AGAS) includes all localities the current application is using. Global identifiers ease to locate objects in the network. Having this global address space furthermore allows to move objects across localities without changing its address.

Instrumentation and Adaptivity includes several performance countering and monitoring tools. They might be helpful when debugging or analyzing the program and offer insights without using additional tools.

Threading Subsystem uses a work-queue based execution strategy to schedule HPX threads. These are either created locally or the threading subsystem converts parcels on receive into HPX threads. In contrast to other systems the ParalleX model maps m internal threads instead of just 1 onto n kernel threads. This allows ParalleX to switch between threads without the need of kernel calls and their context switch overhead.

Local Control Objects (LCO) are used for scheduling and abstract mechanism used to do so. For example, *futures* are proxies for results not yet known and can only be accessed after the result is available. In case the future is accessed earlier, the thread doing so is suspended. Additional to special objects such as futures, HPX also offers traditional concurrency control mechanisms, for instance mutexes and semaphores.

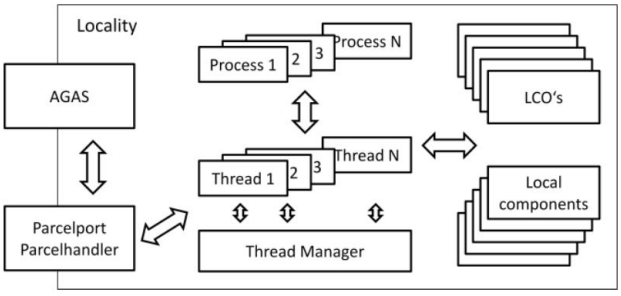

Fig. 1. ParalleX Components Implemented in HPX [10]

2) Tasks in HPX: Various program models increase their efforts on tasking in recent years and ~~increase~~ the possibilities

in local parallelism. However, in contrast to the other program models, HPX offers a way for homogeneous execution of local and remote operations. This is achieved by using the task model combined with a so called *Active Global Address Space*. As all nodes of an application share a common address space, objects and tasks can be migrated across nodes. Instead of moving the data to nodes the tasks can be moved to the data, which allows faster communication as the instructions are often smaller in size than the actual data. Furthermore, the migration can be done without changing the global address as all nodes share the address space. Additional work is saved due to this fact. [9]

Another advantage of HPX which also makes it convenient to C++/C programmers is that it uses the same tasking methods as the C++ standard. No new semantic or syntax is used. However, the actual standard is extended to support remote operations. To use tasking in HPX, tasks have to be defined by `async` and `future`. `async` launches a task asynchronously. `future<T>` represents a value of type `T` which will be available in the future, when the execution of a task finishes. It acts as a placeholder of a result not known yet and when created spawns a new HPX-Thread which is placed on the thread queue. In case another thread wants to access a future it is blocked and has to wait until the value of the future is available. Using `get()` is an explicit option for the programmer to let the thread wait for the future. [9] [2]

The HPX thread scheduling system uses lightweight threads which allow faster context switches and smaller stacks. Task migration is therefore more efficient as less data has to be moved. Furthermore, it is designed to handle millions of tasks efficiently. In [9] HPX tasks are compared to OpenMP tasks for their GFLOPS on various numbers of cores. However the HTTS benchmark is used which uses no-op tasks and can therefore only compare the thread scheduling performance without actual workload. One reason why HPX performed better in the HTTS benchmark is that it uses constraint based synchronization instead of global barriers, like in OpenMP. For example, OpenMP uses implicit barriers at the end of each loop. In contrast to that HPX allows to execute the code following the loop in case no data dependencies are given.

D. Benchmark Algorithms

The authors of [11] introduce a benchmark suite to test the impact of different implementation decisions using OpenMP tasks. The benchmark suite is called *Barcelona OpenMP Task Suite (BOTS)* and includes nine benchmarks:

Alignment is an algorithm aligning protein sequences against every other sequence.

Fourier Transformation tries to approximate a periodic function by several subfunctions.

Fibonacci calculates the n^{th} number of the Fibonacci sequence.

Floorplan computes the optimal floorplan distribution of a number of cells.

Health simulates the Colombian Health Care System.

N Queens tries to find placements of n queens on a chessboard under special condition.

Sort is a special kind of parallel merge sort execution.

SparseLU calculates a LU matrix factorization over sparse matrices.

Strassen hierarchically decomposes a matrix for multiplication of large dense matrices.

The suite offers different versions of each benchmark, e.g. a version of Fibonacci using a cut-off to avoid a high amount of tasks.

E. hpxMP

A rather new approach is *hpxMP*. It's an implementation of the OpenMP standard using an underlying hpx system and introduced in [12]. The motivation of this approach is to ease the migration from OpenMP to Asynchronous Many-Task (AMT) systems. AMT as a new parallel programming paradigm utilizes fine grained tasks to distribute the workload across multiple nodes. It is similar to the well known *MPI+X* approach in which the Message Passing Interface (MPI) is used together with OpenMP for example. However, for certain applications MPI+X offers less scalability and parallel efficiency compared to AMTs, for instance in [13]. HPX as an example for an AMT might also be more convenient as it follows the newest C++ standard and therefore has a more flatten learning curve for C++ programmers. Another advantage of hpxMP might have in contrast to MPI+X is that it uses HPX lightweight threads instead of system threads. They are briefly explained in II-C2.

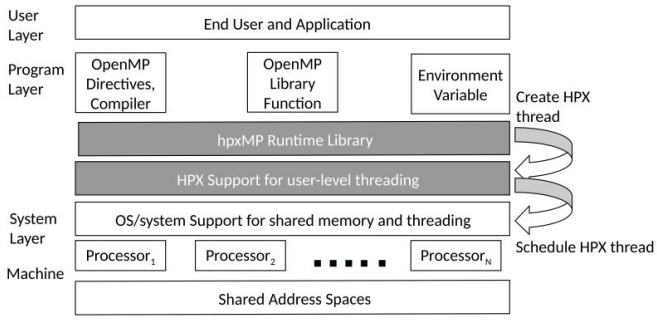

Fig. 2. Adaption of an OpenMP Application to use hpxMP [12]

Figure 2 shows how hpxMP can be used to run OpenMP applications. The application is compiled with OpenMP flags, but linked against the hpxMP runtime library being the first grey layer in the figure. Instead of calling the OpenMP functions, equivalent hpxMP functions are called. These redirect the calls to appropriate functions of the underlying HPX system being the next grey layer in the figure. This system allows to execute OpenMP applications using HPX according to the OpenMP standard.

The authors of [3] furthermore extended hpxMP with further task features of the OpenMP 5.0 standard. Additionally, they compared the new hpxMP system with different OpenMP implementations. Among other benchmarks, they used the BOTS merge sort algorithm. The cut off option is used to

decrease the number of tasks spawned. The results show that hpxMP can already reach similar or better performance results for larger input sizes. This is due to the fact that joining a large number of tasks after a parallel region produce less overhead using HPX threads. Compared to conventional operating system threads HPX threads are lightweight and need less effort when joining. However for smaller input sizes hpxMP reaches slower performance. The authors explain this by the overhead which is introduced by the HPX scheduler. Conventional threads spawned by OpenMP may not be suspended and therefore need to be scheduled less often.

III. Implementation & Design

For this paper the Fibonacci and the Sort algorithm from BOTS are chosen to be implemented, as introduced in subsection II-D. Additionally, a generic algorithm is implemented. All three are implemented using OpenMP, HPX and in a sequential way.

a) Fibonacci: The algorithm can spawn a high amount of tasks with small workload. The workload of forking and joining new threads might have the biggest impact when it comes to performance.

The implementation is done without cut offs for the HPX and OpenMP versions. This means that every recursive call of the Fibonacci function creates a new task.

Listing 1
FIBONACCI OPENMP IMPLEMENTATION

```
long long fibonacci(long long input) {
    if (input < 2 ) return input;
    long long x, y;
    #pragma omp task shared(x) firstprivate(input)
    x = fibonacci(input - 1);
    #pragma omp task shared(y) firstprivate(input)
    y = fibonacci(input - 2);
    #pragma omp taskwait
    return x + y;
}
```

Listing 1 shows a first implementation of the Fibonacci algorithm using OpenMP. It shows which directives are used. As explained in subsection II-B, the `#pragma omp task` directive creates a new task which may be executed. Using the `shared` and `firstprivate` clauses adjusts the variable scopes so that each task has its own instance of the parameter input. Variables `x` and `y` are shared which means no new instance is created. At the end the `#pragma omp taskwait` directive tells the function to wait until all the child tasks finished their execution.

Listing 2
FIBONACCI HPX IMPLEMENTATION

```
long long fibonacci(long long input) {
  if (input < 2) return input;
  hpx::future<long long> n1 =
      hpx::async(fibonacci, input - 1);
  long long n2 = fibonacci(input - 2);
  return n1.get() + n2;
}
```

The HPX implementation of the Fibonacci algorithm can be seen in listing 2. In contrast to OpenMP, HPX works with futures to abstract results of function calls. Calling a function with `hpx::async` creates a task and immediately returns a future to continue the execution. Calling `get` on a future suspends the current thread until this future is returned.

b) Merge Sort: The Sort algorithm of BOTS is slightly adjusted to a normal merge sort. It is a suitable use case, easy to implement and can also spawn a high number of tasks. Similar to Fibonacci the merge sort is also implemented without cut off.

c) Generic Algorithm: The aim of this algorithm is to enable task size adjustment and allow to define the number of tasks easily by parameters.

The algorithm uses two arrays which sizes are defined at the building process. First of all one array is filled by randomized floating point numbers. To compare each run a deterministic seed for the random numbers is chosen. In each turn the element of a vector is equal to a calculation on the element of the other array as it can be seen in a) of figure 3. The sinus function is calculated on the element before it is multiplied by ten and adjusted to an absolute value. This turns are repeated a defined number of times.

The task size can be adapted by parameters and defines how many array elements are calculated by a task. The task size furthermore defines how many tasks are used per run. This number is equal to the array size divided by the task size. As each turn depends on the execution of the previous turn, the number of dependencies can be increased by using more tasks per turn.

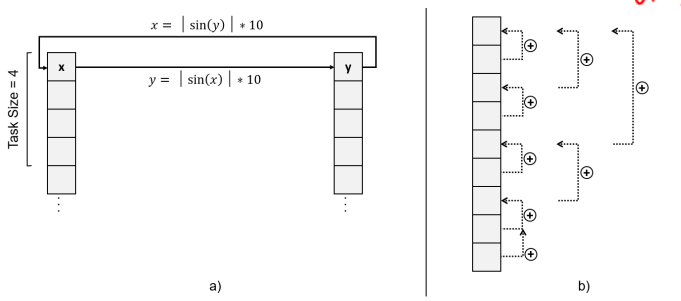

Fig. 3. Generic Algorithm a) Turns b) Summation

After the last turn is executed, all elements of the last calculated array are summed up. This is done by building pairs and adding the value of the second member to the value of the first one and storing the results in the first value's element. Part b of figure 3 illustrates how these pairs are arranged. It also shows how elements are handled that cannot be part of a pair. These leftover element is added to the result of the last pair. This process is continued with the resulting values until only one element is left.

The code can be found on GitHub [14]. It is mainly written in C++, but also contains CMake files and scripts for building, execution, and formatting the results.

TABLE I
HARDWARE SPECIFICATION OF THE ENVIRONMENTS

	CPU	RAM	Cores
Environment 1	Intel Xeon Phi 7210	~ 92 GB	64
Environment 2	AMD EPYC 7551P	~ 124 GB	32
Environment 3	AMD EPYC 7742	~ 3.8 GB	64
Environment 4	AMD Ryzen 5 1600X	~ 9.6 GB	6

TABLE II
SOFTWARE SPECIFICATION OF THE ENVIRONMENTS

	Environment 1	Options
Compiler	GCC 9.2.0	
HPX Version	1.4.0	cxxstd=14 networking=none
Boost Version	1.70.0	cxxstd=14
OpenMP Version	4.5	

IV. ENVIRONMENT

The code is executed on different systems which differ in hardware configuration. This might show different behavior of the implementations in different contexts.

The hardware specifications of the three environments can be seen in table I. Environment 1 has has an Intel Xeon Phi 7210 processor with 64 cores and the available memory is about 92 GB. Environment 2 contains about 124 GB and an AMD EPYC 7551P processor which has a number of cores equal to 32. The third environment owns a 64 cored AMD EPYC 7742 processor. The available memory is quite small compared to the other environments, but sufficient. It has a size of about 3.8 GB.

The software specifications of the environments can be seen in table II. On all systems the same software versions are used and contain `GCC`, `HPX`, `Boost` and `OpenMP`. HPX and Boost are installed and build by using Spack [15] which is a package management tool.

V. RESULTS & DISCUSSION

The results of the three algorithms in this section are created on environment three. They are illustrated using boxplots showing the execution time of 100 runs.

a) Fibonacci: Figure 4 shows the execution time of the 22nd Fibonacci number. It illustrates the times of the sequential, OpenMP and HPX version. The sequential and OpenMP implementation need the fewest time to complete, whereas the HPX version takes a lot more time.

b) Merge Sort: The measurements of figure 5 show the execution times of merge sort. The algorithm is run on 10.000 random elements which are created in a deterministic way. Similar to Fibonacci, merge sort also creates a lot of tasks and HPX shows the slowest execution time. Sequential has the shortest average time and the OpenMP average times differ more significantly this time compared to the Fibonacci example.

c) Generic Algorithm: Figure 6 shows the measured execution times for the generic algorithm in his three versions. 20 turns are made with a task size of 20 and the array size is 1,048,576. It can be seen that the range of OpenMP times

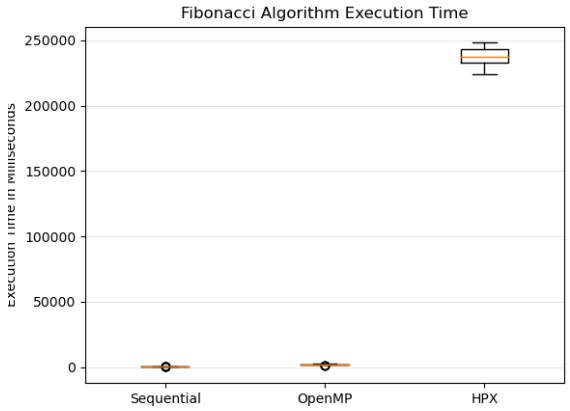

Fig. 4. Execution Times of the Fibonacci Algorithm (Environment 3, 64 Threads)

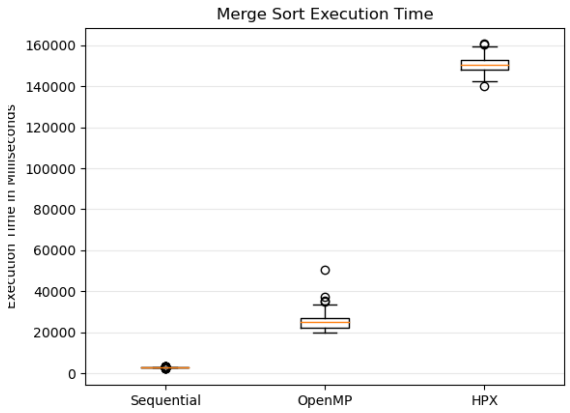

Fig. 5. Execution times of Merge Sort (Environment 3, 64 Threads)

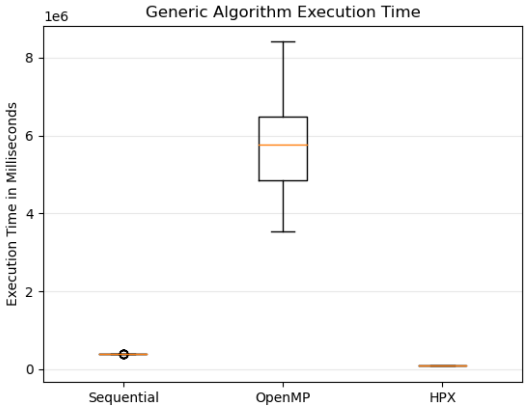

Fig. 6. Execution times of the generic algorithm (Environment 3, 64 Threads)

TABLE III
EXECUTION TIMES OF THE FIBONACCI ALGORITHM WITH CUTOFFS IN MILLISECONDS

	With Cutoff	Without Cutoff
OpenMP	2057.34 ms	394.87 ms
HPX	237491.49 ms	5267.67 ms

is quite big compared to the sequential and HPX version. Additionally, the OpenMP version takes the longest time to complete the execution whereas HPX is the fastest of these three.

A. Discussion

It can be seen that HPX has some issues keeping up with the sequential and OpenMP implementation of Fibonacci and merge sort. Both algorithms spawn a lot of tasks in hierarchical structure. This means each task, except of the root task, has a parent task. This parent task has to suspend its execution until its child tasks finished their execution. In contrast to threads in OpenMP HPX threads may be suspended which results in scheduling tasks occurring more often. This leads to a bigger scheduling workload for HPX.

The generic algorithm however uses a different structure as it spawns various tasks for an array which need to be finished before spawning the next tasks for the next turn. In this concept HPX can use its strength of lightweight threads. These fork and join faster and with less work to do.

VI. OPTIMIZATION

A. Fibonacci - Cutoff

One possible optimization used in the BOTS benchmarks is implementing a cutoff option. This can be achieved by using if clause which is provided by the task directive as it is described in [16]. In case this clause evaluates to false no task will be spawned and the execution continues with the function call in a normal sequential way. The cutoff value is used to specify at which level the tasking spawning has to be stopped. A level parameter is added to the Fibonacci function and increased every recursive call. In case the level parameter is equal or greater than the cutoff value, the Fibonacci algorithm will be continued sequentially without spawning tasks. Table III shows the results without a cutoff and with a cutoff level equal to eight. The mean values of 100 runs are 2057.34 ms and 394.87 ms for OpenMP without and with a cutoff. The average measurements for HPX are 237491.49 ms without and 5267.67 ms with cutoff. It can be seen that the HPX implementation benefits more by the cutoff. HPX speeds up by a ratio of approximately 45 and the OpenMP implementation speeds up only by 5.

B. OpenMP - untied / tied Tasks

Another optimization approach might be to see if there is a difference in using tied or untied tasks. Untied tasks improve the load balancing as each free thread can start executing a task which is ready to be executed. The disadvantage of untied tasks is that the data is not always available locally. This means in case a thread wants to execute a ready task which is not created by it, the data and context has to be transferred to new execution location.

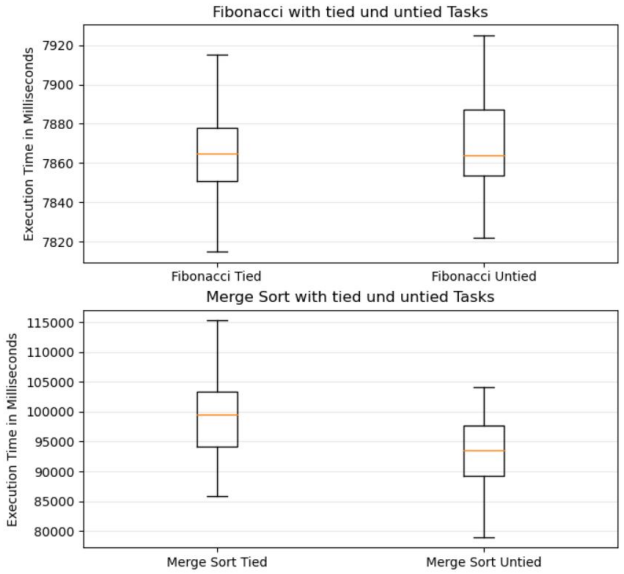

Fig. 7. Execution times of Fibonacci and Merge Sort with tied and untied Tasks in OpenMP (Environment 1, 256 Threads)

This experiment is run on environment 1 calculating again the Fibonacci number of 22. As figure 7 shows there is no significant difference in execution time for both algorithms. Using tied or united tasks may only have minor impacts. As already mentioned this is due to the trade of of data locality and load balancing.

C. HPX - Fibonacci Dataflow

HPX provides further LCOs than just futures. Another object which might be useful is the *dataflow* object. It is an object which is used to define dependencies in functions easily. Dataflows take a function and its parameters when initialized and call the function when all of its parameters are available. This might be the case for example if parameters are returned by futures. However, it is also possible to concatenate dataflow objects to create a dependent hierarchy. [2]

This experiment compares the HPX Fibonacci implementation using two futures with an implementation utilizing dataflows and another Fibonacci implementation using `when_all`. The last approach follows the same concept as the dataflow implementation, but works without this object type. `when_all` takes futures as parameters and starts executing when the values of all futures are available. For this implementation the example code of HPX is taken as reference. [17] The idea is to utilize the dataflow principle to avoid blocked tasks and increase the scheduling overhead.

The average execution times of 100 runs on environment 3 for the 22nd Fibonacci number's calculation are illustrated in figure 8. It can be seen that the normal Fibonacci implementation has the shortest execution times of these three.

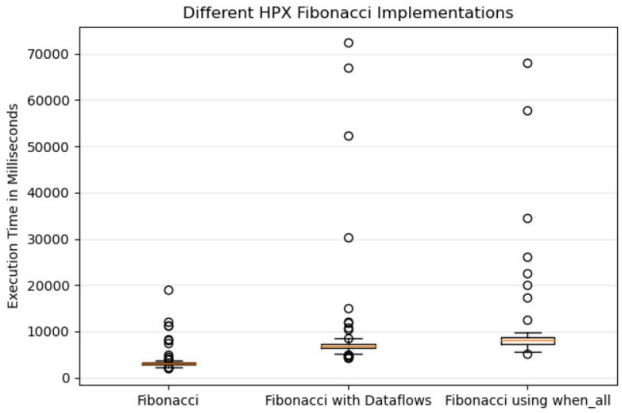

Fig. 8. Execution times of Fibonacci using HPX and different dataflow constructs (Environment 3, 64 Threads)

This means that the dataflow principle cannot help avoiding scheduling overhead for HPX implementations. Instead it produces additional overhead.

D. HPX - Thread Scheduling Policies

The authors of [12] and [2] state different thread scheduling policies in HPX. However, not all of them are available at the HPX version used. Therefore only the policies used are explained. The experiment is run on environment 2.

local-priority gives each operating system (OS) thread one queue they can pull their work from. Furthermore a high and low priority queue are created which is accessible by each OS thread. Work from these queues is only pulled in case the dedicated OS thread queue is empty. In doing so the high priority queue's work is taken first. Additionally, the local-priority policy is available in first-in-first-out (fifo) and last-in-first-out (lifo) version.

local only maintains one queue per OS thread the work is put to and pulled from.

abp-priority is similar to the local-priority policy. There are also a global high priority and low priority queue. However each OS threads owns a double ended lock free queue. Elements can be added and removed on both sides of this type of queue. Furthermore the abp-priority policy can also be used in fifo and lifo version.

static has one queue per OS thread and tasks are distributed by round robin. In this policy no work stealing is available.

static-priority is similar to local-priority as there are queues for each OS thread, high priority queues and low priority queues which can be accessed globally. In contrast to the local-priority policy the tasks are distributed by round robin and no work stealing is available.

The figures show the execution time of the three implemented algorithms with the previously explained thread scheduling policies. The parameters are the same as stated in section III. Figure 9 is a boxplot diagram of the generic algorithm, figure 10 shows the execution times of the Fibonacci algorithm and figure 11 illustrates the execution times of merge sort. It can be seen that the local policy accelerates

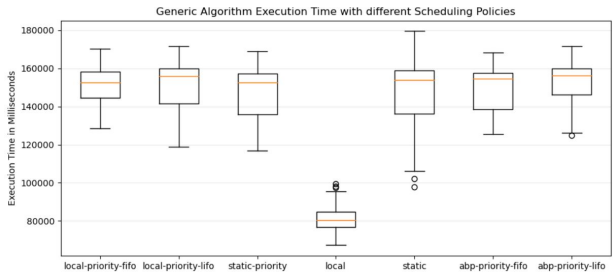

Fig. 9. Execution times of the generic algorithm using different scheduling policies(Environment 2, 64 Threads)

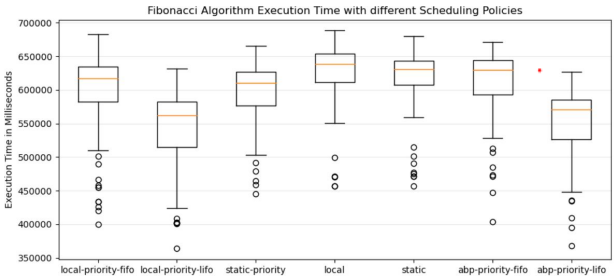

Fig. 10. Execution times of the Fibonacci algorithm using different scheduling policies (Environment 2, 64 Threads)

the generic algorithm execution significantly compared to the other policies. This may be because it produces little overhead as it does not have to maintain priority queues. The significant difference to the static scheduling policy is that, when using the local policy, tasks are not distributed via round robin and task stealing is allowed. These facts may lead to a better load balancing although the task size does not vary much. The other two algorithms show no significant different in the policies. However a trend can be seen as the mean values of the lifo policies show a shorter execution time in all cases. This may be due to the fact that Fibonacci and merge sort create a task hierarchy in which the early created tasks are suspended till the end.

E. Generic Algorithm - Number of Tasks

For a better comparison of the three algorithms, various task sizes might be used. The generic algorithm allows to adjust task sizes by parameters. The second environment is taken for this test run in which tasks sizes ranging from 10 elements to 500 elements per task. The total number of elements in the arrays $1,048,576$ and the arrays are computed 50 times.

Figure 12 illustrates the execution time for all three elements. The mean value of 50 measurements is taken. It can be seen that the sequential and HPX implementation are quite constant. In contrast, OpenMP takes more time when the task size is rather small and reaches the execution time of the sequential version at a task size of 410. If the task size is bigger OpenMP takes less time compared to the sequential implementation. However, it cannot reach the times of the HPX implementation. Figure 13 shows the same data however only the task sizes greater and equal to 300. Especially, the

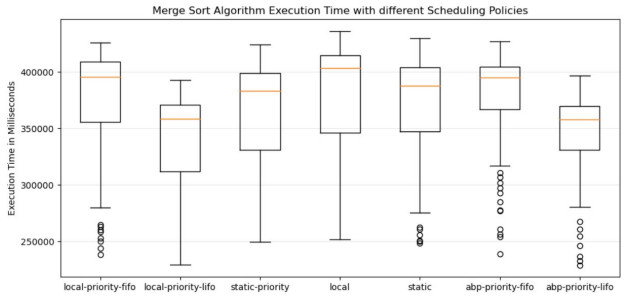

Fig. 11. Execution times of Merge Sort algorithm using different scheduling policies (Environment 2, 64 Threads)

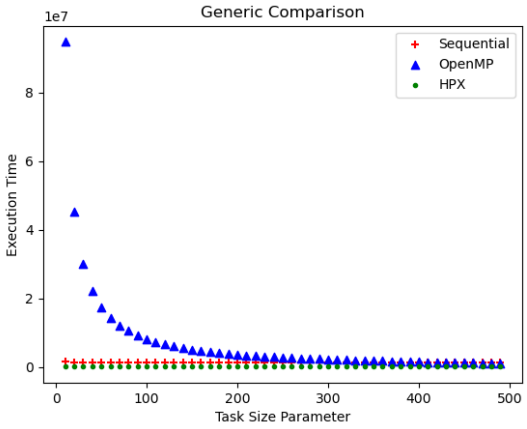

Fig. 12. Execution times of generic algorithm with various task sizes (Environment 2, 64 Threads)

trends of OpenMP and the sequential version can be seen in detail. Important to mention is that changing the task size in the generic algorithm does not increase the overall workload. This can be seen at the sequential implementation. Changing the task size does only increase or decrease the tasks which are created. In contrast to the Fibonacci algorithm and merge sort, the generic algorithm does not create hierarchical dependencies. Task synchronization is done when all elements of an array are calculated. HPX benefits from that structure as the lightweight threads used produce less overhead when forking and joining. The scheduling overhead is smaller than the benefit from the parallel execution and therefore HPX becomes faster than the sequential implementation. The joining and forking overhead can be seen in the execution times of OpenMP. OpenMP does have operating system threads and therefore produces more overhead having bigger amounts of tasks. However, reaching a certain task size OpenMP also takes less time than the sequential execution of the generic algorithm.

F. Generic Algorithm - Task Sizes

As the previous example of the generic algorithm does only increase the amount of tasks used and not the overall workload, another approach is also implemented. In this experiment the

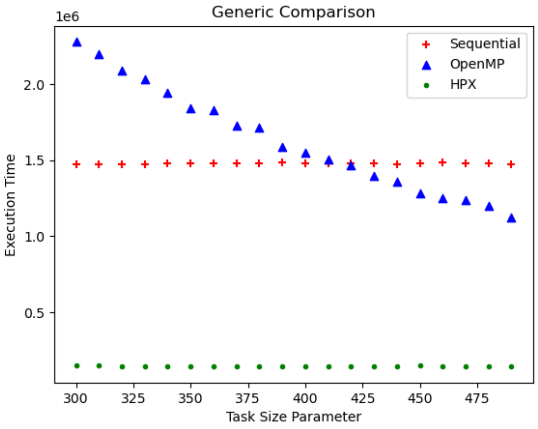

Fig. 13. Execution times of generic algorithm with biggest task sizes (Environment 2, 64 Threads)

generic algorithm is adjusted to produce a squared workload per array element. The workload can be adjusted by a parameter and varies in this experiment. In contrast to the experiments before, each task does only contain one element. This setup does allow to have a constant number of tasks with an option to increase or decrease the task size and overall workload. In each task the elements value is multiplied a defined amount of times. The parameter for defining this size is squared, which means that if the parameter is equal to 3 the element is multiplied 9 times. Each time the value is multiplied by the number of that multiply calculation. In the example before the first value will be multiplied by 1 and multiplied by 9 in the last calculation. The sinus function is then applied on the results and all of them are summed up and copied to the next array. This experiment is also run on environment two having the same array size as the experiment before. The number of calculations starts with 1x1 and increases by 2 in each dimension. The biggest task size reached is 21x21. For the HPX version the local thread scheduling policy is used.

Figure 14 part a) shows the average execution time of 25 runs. As already mentioned, the number of tasks are the same for all executions. Increasing the matrix size increases the task size and overall workload as it can be seen by the sequential execution's curve. The HPX execution time looks like to have a quite constant trend. However, this is due to the graphical illustration. Having only the HPX data points in part b) of figure 14 one can see that the execution time of HPX also increases. The numbers are just so small that it cannot be seen in the scatter plot above.

OpenMP's curve decreases with an increasing task size, although the overall workload increases. This behavior is quite strange as the number of tasks stays constant. Running the same experiment with setting `OMP_NUM_THREADS=1` the execution times are e

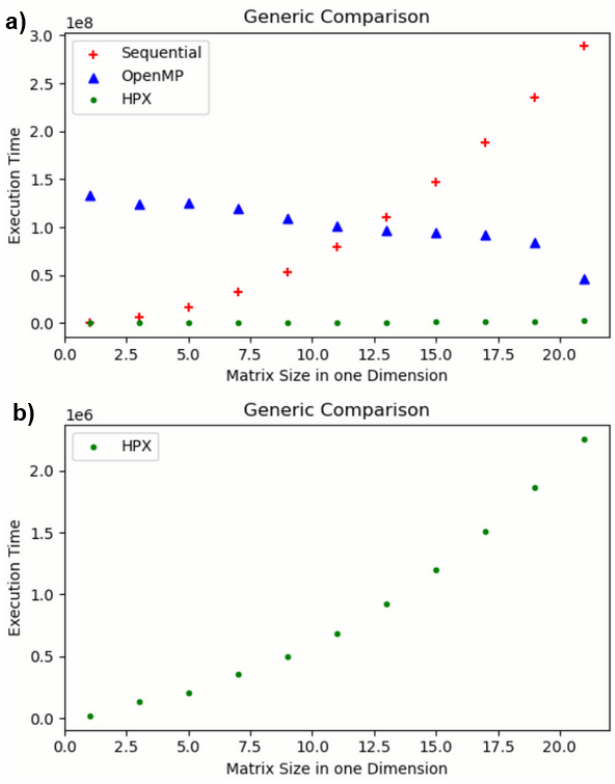

Fig. 14. Execution Times of Generic Algorithm Task Matrices of all Versions and only HPX(Environment 2, 64 Threads)

G. Generic Algorithm - Sanity Tests

The last experiment of this section is a sanity test to test the behavior of the OpenMP and HPX implementations using one and two tasks in parallel. It is expected that when executing the algorithm with a tasks size equal to the size of the array the execution time must be similar to the one of the sequential implementation. The generic algorithm is given a task size equal to the size of the array and another one equal to half of the array size. Therefore only one or two tasks are in use per turn. Furthermore the final summation of all elements is exchanged in favor of the sequential version.

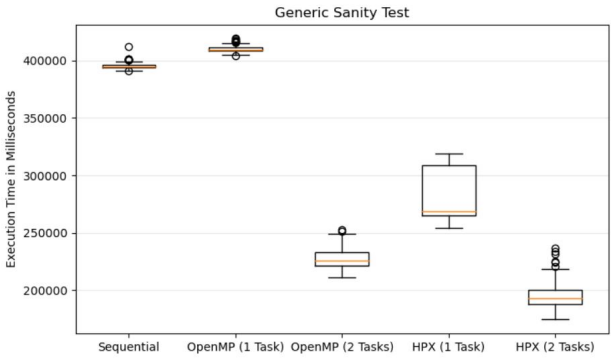

Fig. 15. Sanity Tests for Generic Algorithm (Environment 3, 64 Threads)

Figure 15 shows the execution times for this experiment. It can be seen that the OpenMP version using one tasks takes about the same time as the sequential version and using two tasks decreases the execution time significantly. For the OpenMP implementation it is important to set the `OMP_NUM_THREADS` variable equal to the simultaneously used tasks. Otherwise the OpenMP version takes longer than the sequential implementation. These may be due to the fact that different cores may execute the code. Doing so forces to use a different part of the CPU cache and nihilists the lines loaded before.

Important to notice is that the HPX implementation takes less time to execute the generic algorithm with one task. This is also tested using *system* and *tcalloc* allocation options during build. However both options result in no significant difference in execution time.

VII. Conclusion & Future Work

hpxMP is currently in the process of development and soon allows to decide to use the OpenMP or HPX thread system with the HPX programming interface. The findings of this paper might be used to help a programmer decide one of these thread systems. This paper compared both systems for their tasking performance using three algorithms, Fibonacci, Merge Sort and a generic one. It came clear that HPX achieves minor performance with a high amount of tasks in hierarchical structure compared to OpenMP. This is due to the fact that HPX produces more overhead when it comes to scheduling. In contrast to OpenMP, HPX threads may be suspended which leads to the situation that scheduling is conducted more often. The implemented Fibonacci using the cutoff option undermines that statement as HPX benefits much more from this compared to OpenMP. However, looking at the experiment using the generic algorithm with variable number of tasks and task sizes, it can be seen that this lack of performance for HPX is only the case for hierarchical task spawning, for instance in Fibonacci and Merge Sort. In the experiments using the generic algorithm HPX could show that its advantage of lightweight threads. They produce less overhead when forking and joining and let HPX perform best for that algorithm. The paper furthermore analyzed optimizations for HPX, e.g. using dataflow principles or different scheduling policies. These experiments did not really lead to any significant gain in performance, however showing slight trends. For example the local thread scheduling policy for the generic algorithm.

A remaining open field concerning both run time systems is a comparison between their GPU support. As HPX and OpenMP may move computations to these accelerators it might be interesting to see how both perform utilizing these with tasks. Furthermore, experiments are yet only done using these three algorithms. Both systems should be further tested on algorithms showing a different behavior concerning task sizes, number of tasks and task hierarchy.

References

[1] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. The design of

openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[2] The STEllAR Group. Hpx documentation, 2020. stellar-group.github.io/hpx-docs/latest/html/index.html, accessed March 18th 2020.

[3] Tianyi Zhang, Shahrzad Shirzad, Bibek Wagle, Adrian S. Lemoine, Patrick Diehl, and Hartmut Kaiser. Supporting openmp 5.0 tasks in hpxmp – a study of an openmp implementation within task based runtime systems.

[4] Ahmad Qawasmeh, Abid M. Malik, and Barbara M. Chapman. Openmp task scheduling analysis via openmp runtime api and tool visualization. In *IEEE International Parallel & Distributed Processing Symposium workshops (IPDPSW), 2014*, pages 1049–1058, Piscataway, NJ, 2014. IEEE.

[5] Soumen Chakrabarti and Katherine Yelick. Implementing an irregular application on a distributed memory multiprocessor. *ACM SIGPLAN Notices*, 28(7):169–178, 1993.

[6] James LaGrone, Ayodunni Aribuki, Cody Addison, and Barbara Chapman. A runtime implementation of openmp tasks. In Barbara Chapman, editor, *OpenMP in the petascale era*, volume 6665 of *Lecture notes in computer science, 0302-9743*, pages 165–178. Springer, Heidelberg, 2011.

[7] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of openmp task scheduling strategies. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *OpenMP in a new era of parallelism*, volume 5004 of *Lecture notes in computer science, 0302-9743*, pages 100–110. Springer-Verlag, Berlin, 2008.

[8] Karl Fürlinger and David Skinner. Performance profiling for openmp tasks. In Matthias S. Müller, Bronis R. de Supinski, and Barbara Chapman, editors, *Evolving OpenMP in an age of extreme parallelism*, volume 5568 of *Lecture notes in computer science, 0302-9743*, pages 132–139. Springer, Berlin, 2009.

[9] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx - a task based programming model in a global address space. In Allen D. Malony and Jeff Hammond, editors, *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models - PGAS '14*, pages 1–11, New York, New York, USA, 2014. ACM Press.

[10] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In Leonard Barolli, editor, *International Conference on Parallel Processing workshops, 2009*, pages 394–401, Piscataway, NJ, 2009. IEEE.

[11] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *International Conference on Parallel Processing, 2009*, pages 124–131, Piscataway, NJ, 2009. IEEE.

[12] Tianyi Zhang, Shahrzad Shirzad, Patrick Diehl, R. Tohid, Weile Wei, and Hartmut Kaiser. An introduction to hpxmp. In Andrei Poenaru, editor, *Proceedings of the International Workshop on OpenCL*, ICPS: ACM international conference proceeding series, pages 1–10, New York, NY, USA, 2019. ACM.

[13] Gregor Daiß, Parsa Amini, John Biddiscombe, Patrick Diehl, Juhan Frank, Kevin Huck, Hartmut Kaiser, Dominic Marcello, David Pfander, and Dirk Pfüger. From piz daint to the stars. In Michela Taufer, Pavan Balaji, and Antonio J. Peña, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–37, New York, NY, USA, 11172019. ACM.

[14] Tobias Schiffmann. Code implementation, 2020. github.com/t0BEE/Task-Based-Analysis, accessed August 20th 2020.

[15] Spack. Spack documentation, 2020. spack.readthedocs.io/en/latest/index.html, accessed August 7th 2020.

[16] M Klemm and B Supinksi. *OpenMP Application Programming Interface Specification Version 5.0*. 2018.

[17] The STEllAR Group. Hpx example implementations, 2020. github.com/STEllAR-GROUP/hpx/tree/master/examples/quickstart, accessed August 15th 2020.