Using HPX and OP2 for Improving Parallel Scaling Performance of Unstructured Grid Applications

Zahra Khatami$^{1,2,3}$, Hartmut Kaiser$^{1,2,4}$, and J. Ramanujam$^{1,5}$
$^1$Center for Computation and Technology, Louisiana State University
$^2$The STE|AR Group, http://stellar-group.org
$^3$zkhatami@lsu.edu, $^4$hkaiser@cct.lsu.edu, $^5$ram@cct.lsu.edu
Baton Rouge, Louisiana, USA

Abstract—Computer scientists and programmers face the difficulty of improving the scalability of their applications while using conventional programming techniques only. As a baseline hypothesis of this paper, we assume that an advanced runtime system can be used to take advantage of the available parallel resources of a machine in order to achieve the highest parallelism possible. In this paper, we present the capabilities of HPX—a distributed runtime system for parallel applications of any scale—to achieve the best possible scalability through asynchronous task execution [1].

OP2 is an active library which provides a framework for the parallel execution for unstructured grid applications on different multi-core/many-core hardware architectures [2]. OP2 generates code which uses OpenMP for loop parallelization within an application code for both single-threaded and multi-threaded machines. In this work, we modify the OP2 code generator to target HPX instead of OpenMP, i.e. port the parallel simulation backend of OP2 to utilize HPX. We compare the performance results of the different parallelization methods using HPX and OpenMP for loop parallelization within the Airfoil application. The results of strong scaling and weak scaling tests for the Airfoil application on one node with up to 32 threads are presented. Using HPX for parallelization of OP2 gives an improvement in performance by 5%-21%.

By modifying the OP2 code generator to use HPX’s parallel algorithms, we observe scaling improvements by about 5% as compared to OpenMP. To fully exploit the potential of HPX, we adapted the OP2 API to expose a future and dataflow-based programming model and applied this technique for parallelizing the same Airfoil application. We show that the dataflow-oriented programming model, which automatically creates an execution tree representing the algorithmic data dependencies of our application, improves the overall scaling results by about 21% compared to OpenMP. Our results show the advantage of using the asynchronous programming model implemented by HPX.

Index Terms—High Performance Computing, HPX, OP2, Asynchronous Task Execution.

I. INTRODUCTION

Today, achieving adequate parallel application scalability is one of the major challenges for everyday programmers, especially when using conventional programming techniques [3], [4]. To achieve efficient utilization of compute resources, it is needed to take full advantage of all available parallel resources. Using a programming model can help to overcome this challenge and enables significant improvements of levels of parallelism by intrinsically avoiding resource starvation, hiding latencies, and as a result reducing overheads. Parallelization is often implemented by decomposing the domain space into several sub-domains and assigning each of them to a group of the processors. However, the overhead time imposed due to the required communication between processors may inhibit the application’s scalability. As a results, in addition to space, time should be considered as a factor helping to get a maximum possible parallelism level [5], [6]. So the parallelization should be done in both space and time domains.

HPX [7] helps overcoming these difficulties by exposing a programming model which intrinsically reduces the SLOW factors [8]. The SLOW factors are the main reasons for reduced parallel scalability due to a) poor utilization of resources caused by lack of available work (Starvation); b) the time-distance delay of accessing remote resources (Latencies); c) the cost for managing parallel actions (Overhead); d) the cost imposed by oversubscription of shared resources (Wait). HPX helps the programmer to parallelize his code to achieve the best possible scalability through asynchronous task execution, which results in enabling parallelization in both space and time domains. As a result, it removes global barrier synchronization and improves parallel application performance. In this paper, HPX is used to improve the performance of the applications’ codes generated by OP2 that result in increasing the parallel application scalability and performance.

OP2 provides a framework for the parallel execution of unstructured grid applications [2]. It’s design and development is presented in [12], [13]. With OP2, applications can be targeted to execute on different multi-core/many-core hardware [12], [14]. To achieve further performance gains with OP2 on modern multi-core/many-core hardware, some optimizations should be applied to improve performance for different parallel applications. This can be obtained by avoiding the SLOW factors as much as possible. Originally, OpenMP is used for loop parallelization in OP2 on a single node and on distributed nodes, where it is used in conjunction with MPI. However, the #pragma omp parallel for used for loop paralleliza-
tion in OP2 causes an implicit global barrier at the end of loop, which prevents achieving an adequate application speedup. The use of HPX’s parallelization methods instead of OpenMP helps eliminating these SLOW factors and enables extracting more parallelism for the parallel applications.

This paper describes the results from porting the parallel simulation backend of OP2 to utilize HPX. We study scalability and performance of Airfoil application when targeting the OP2 code generator for OpenMP and HPX. We compare the achievable scalability and performance of the HPX backend when using the original (unchanged) API of OP2 and an adapted version of the OP2 API. The analysis is performed using a standard unstructured mesh finite volume computational fluid dynamics (CFD) code, presented in [17], which has both direct and indirect loops. We demonstrate OP2 loops in Figure 2, which includes op_par_loop_save_soln and op_par_loop_adt_calc loops from Airfoil: op_par_loop_save_soln is a direct loop that applies save_soln on cells based on the p_q and p_gold arguments, and op_par_loop_adt_calc is an indirect loop that applies op_par_loop_adt_calc on cells based on p_x, p_q, and p_adt arguments passed to the loop. The function op_arg_dat creates an OP2 argument based on the information passed to it. These arguments passed to loops in OP2 explicitly indicate that how each of the underlying data can be accessed inside a loop: OP_READ (read only), OP_WRITE (write) or OP_INC (increment to avoid race conditions due to indirect data access) [2]. For example, in op_arg_dat(p_x,0,pcell,2,"double",OP_READ) from Figure 2, OP_READ marks the data as read only. This argument is created from its inputs, where p_x is the data, 0 indicates that the data is accessed indirectly, pcell is the mapping between the data, 2 is the data dimension, and double is the data type.

OP2 is designed to achieve near-optimal scaling on multicore processors. In [14], [15], it was demonstrated that OP2 is able to produce a near-optimal performance in parallel loops for different frameworks without application programmer intervention. However, starvation, latencies, and communication overheads in parallelization using conventional techniques usually hinders scalability. Most of the conventional parallelization methods are based on the fork-join model, where the computational process is stopped if the results from the

Figure 1: A mesh example that illustrates the code used to describe OP2’s API in Figure 2

data (node) (set 0)

0

1

2

3

4

5

6

7

8

Figure 2: The OP2 API functions op_par_loop_save_soln and op_par_loop_adt_calc represent loops from the Airfoil application. op_par_loop_save_soln creates a direct loop and op_par_loop_adt_calc creates an indirect loop.
previous step are not completed yet. As a result, there is always a (implicit) global barrier after each step.

#pragma omp parallel for is used in the code generated by OP2 for both single-threaded and multi-threaded applications. #pragma omp parallel for has an implicit global barrier that avoids extracting optimal parallelism from a parallel application. In this research, HPX is used for loop parallelization instead of using OpenMP. HPX allows automatic creation of an execution tree for an application, which represents a dependency graph. This enables HPX to asynchronously execute tasks as soon as all data dependencies for this task have been satisfied. The performance of HPX is explained in more details in Section II-B.

Both direct and indirect loops with the Airfoil application are parallelized with OpenMP. An optimization of Airfoil using HPX for parallelizing loops, directly generated by OP2 is discussed in more detail in Section III. The source-to-source code translator for OP2 is written in Matlab and Python [14]. In this research, a Python source-to-source code translator is modified to automatically generate parallel loops with HPX instead of OpenMP.

B. HPX

HPX is a parallel C++ runtime system that facilitates distributed operations and enables fine-grained task parallelism. Using fine-grained tasks results in better load balancing, lower communication overheads, and better system utilization. HPX has been developed to overcome conventional limitations such as global barriers and poor latency hiding [8] by embracing a new way of coordinating parallel execution. It has been developed for different architectures, such as large Non Uniform Memory Access (NUMA) machines, SMP nodes, and systems using Xeon Phi accelerators. Also, HPX uses the light-weight threads, which have extremely short context switching times that results in reducing latencies even for very short operations.

In HPX, asynchronous function execution is the fundamental bases of asynchronous parallelism. HPX’s design focuses on parallelism rather than concurrency, which is defined to have several simultaneously concurrent computations touching the same data, while on the other hand, by parallelism we refer to simultaneous execution of independent tasks [18]. This makes HPX to expose both, time and spatial parallelization [7] due to using future, which enables asynchronous execution of tasks.

A future is a computational result that is initially unknown but becomes available at a later time [19]. The goal of using future is to let every computation proceed as far as possible. Using future enables to continue with the execution without waiting for the results of the previous step to be completed, which eliminates the global barrier at the end of the execution of the parallel loop. future based parallelization provides rich semantics for exploiting higher level parallelism available within each application that may significantly improve scaling. Figure 3 shows the scheme of future performance with 2 localities, where a locality is a collection of processing units (PUs) that have access to the same main memory. It illustrates that other threads do not stop their progress even if the thread, which waits for the value to compute, is suspended. Threads access the future value by performing a future.get(). When the result becomes available, the future resumes all HPX suspended threads waiting for the value. It can be seen that this process eliminates the global barrier synchronizations at the end of application parallelization while only those threads that depend on the future value are suspended. With this scheme, HPX allows asynchronous execution of threads.

III. AIRFOIL APPLICATION WITH HPX

For our evaluation we choose the Airfoil application [17], which uses an unstructured grid and consists of five parallel loops: op_par_loop_save_soln, op_par_loop_adt_calc, op_par_loop_res_calc, op_par_loop_bres_calc, op_par_loop_update, of which the op_par_loop_save_soln and op_par_loop_update loops are direct loops and the others are indirect loops. Figure 4 shows the sequential loops used in the Airfoil application within Airfoil.cpp. Saving old data values, applying the computation on each data value and updating them are implemented within these five loops. Each loop iterates over a specified data set and performs the operations with the user’s kernels defined in a header file for each loop: save_soln.h, adt_calc.h, res_calc.h, bres_calc.h and update.h. Additionally, the OP2 API provides a parallel loop function allowing the computation over sets through op_par_loop for each loop in Figure 4. Figure 5 shows the loop of the op_par_loop_adt_calc function from Figure 4 parsed with OP2, which illustrates how each cell updates its data value by accessing the blockId, offset_b, and nelem.
process can be found in [17]. More details about the Airfoil application and its computation blockId data elements. The value of one. indirect save_soln and data values, applying the computation, and updating each data value.

Figure 5: Five loops are used in Airfoil.cpp for saving old data values, applying the computation, and updating each data value. save_soln and update loops are direct loops and the others are indirect one.

data elements. The value of blockId is defined based on the value of blockIdx captured from the OP2 API. offset_b and nelem are computed based on the value of blockId. The arguments are passed to the adt_calc user kernel subroutine, which does the computation for each iteration in the inner loop from offset_b to offset_b+nelem for each iteration of the outer loop from 0 to nblocks. More details about the Airfoil application and its computation process can be found in [17].

```
#pragma omp parallel for
for(int blockIdx=0; blockIdx<nblocks; blockIdx++) {
    int blockId = //based on the blockIdx in OP2 API
    int nelem = //based on the blockId
    int offset_b = //based on the blockId

    for ( int n=offset_b; n<offset_b+nelem; n++ ) {
        ...
        adt_calc(...);
    }
}
```

Figure 5: #pragma omp parallel for is used for loop parallelization in OP2, for the Airfoil application, on one node and on distributed nodes using MPI.

As shown in Figure 5, #pragma omp parallel for is used for each loop passed with op_par_loop in OP2 for parallelizing the loops on one node and on distributed nodes. However, scalability is limited due to sequential time as described by Amdahl’s Law caused by an implicit barrier between the parallel loops in the fork-join model [20]. HPX parallelization methods are used here instead of OpenMP to achieve optimal parallelization of the loops generated by OP2.

In this research we use two different HPX parallelization methods: A) By modifying the OP2 code generator while using the original OP2 API, and B) By changing the OP2 API to use the HPX future and dataflow methods. The comparison results of these two methods to OpenMP can be found in Section IV.

A. HPX Parallel Algorithms (modifying code generator)

In this Section, we study two different HPX parallelization methods on the Airfoil application by modifying the OP2 code generator to use HPX parallel algorithms while using the original OP2 API. In Section III-A1, parallel::for_each with par as an execution policy is used for parallelizing both direct and indirect loops. In Section III-A2, async and for_each with par is used for parallelizing the direct loops, and for the indirect loops, for_each with par(task) as an execution policy is implemented.

1) for_each: In this method, we implement one of the execution policies of HPX to make the loops shown in Figure 4 execute in parallel. The list of the execution policies can be found in [21] and [1]. par as an execution policy is used while implementing for_each. We modified the OP2 source-to-source translator with Python to automatically produce for_each instead of using #pragma omp parallel for for the loop parallelization. In this method Airfoil.cpp (Figure 4) and the OP2 API remain unchanged.

This example exposes the same disadvantage as the OpenMP implementation, which is the representation of fork-join parallelism that introduces the global barriers at the end of the loop. By using for_each, HPX is able to automatically control the grain size during execution. Grain size is the amount of time a task takes to execute. As discussed in Section II-B, HPX enables fine-grained task parallelism and controls the grain size to distribute tasks to all available threads. Grain size, chunk_size within HPX, is determined by the auto-partitioner algorithm, that estimates the chunk size by sequentially executing 1% of the loop. So for_each helps creating sufficient parallelism by determining the number of iterations to run on each HPX thread that reduces communication overheads [6]. Figure 6 shows the loop of op_par_loop_adt_calc function parsed with OP2.

However, it should be considered that if the computational time of a loop is too large, using an auto-partitioner algorithm within HPX will not be efficient. Since for the large loops, 1% execution time of a loop used for determining a grain size will affect the application’s scalability, HPX provides another way to avoid degrading the scalability while using for_each. Grain size can be specified as a static chunk size with for_each(par.with(scs)) before executing a loop; scs is defined by static_chunk_size scs(SIZE). Figure 7 shows the loop of op_par_loop_adt_calc function with static chunk_size implemented with static_chunk_size scs(SIZE). The experimental result for the Airfoil application is discussed in Section IV.

2) async and for_each: Here, we implement two different parallelization methods for the loops based on their
auto r = boost::irange(0, nblocks);
hpx::parallel::for_each(par,
    r.begin(), r.end(), {[&](std::size_t blockIdx) {

    int blockId = //based on the blockIdx in OP2 API
    int nelem = //based on the blockId
    int offset_b = //based on the blockId

    for ( int n=offset_b; n<offset_b+nelem; n++ ){
        ...
        ...
        adt_calc(...);
    }
});

Figure 6: Implementing for_each for loop parallelization in OP2. HPX is able to control the grain size in this method. As a result, it helps in reducing processor starvation caused by the fork-join barrier at the end of the execution of the parallel loop.

auto r = boost::irange(0, nblocks);
hpx::future<void> new_data;
new_data=hpx::parallel::for_each(par(task), r.begin(), r.end(), {[&](std::size_t blockIdx){

    int blockId = //based on the blockIdx in OP2 API
    int nelem = //based on the blockId
    int offset_b = //based on the blockId

    for ( int n=offset_b; n<offset_b+nelem; n++ ){
        ...
        ...
        adt_calc(...);
    }
});

Figure 7: Implementing for_each for loop parallelization in OP2. HPX is able to avoid degrading the scalability for small loops by defining a static grain size with dynamic_chunk_size scs(SIZE) before the parallel loop execution. types. For the direct loops, async and for_each with par as an execution policy is used. For the indirect loops, for_each with par(task) as an execution policy is implemented. The calls to async and par(task) provide a new future instance, which represents the result of the function execution, making the invocation of the loop asynchronous. Asynchronous task execution means that a new HPX-thread will be scheduled. As a result it eliminates the global barrier synchronization when using for_each in Section III-A1.

In Figure 8, async and for_each with par is used for op_par_loop_save_soln, which is a direct loop and returns a future representing the result of a function. In Figure 9, for_each with par(task) is used for op_par_loop_adt_calc, which is an indirect loop and it also returns a future representing the result of a function. The futures returned from all direct and indirect loops allow asynchronous execution of the loops.

In this method OP2 API is not changed but Airfoil.cpp is changed as shown Figure 10. Each kernel function within op_par_loop returns a future stored in a new_data. Each future depends on a future in a previous step. So, new_data.get() is used to get all futures ready before the next steps. The placement of new_data.get() depends on the application and the programmer should put them manually in correct place by considering the data dependency between loops. In the next Section, we address this problem.

B. HPX with the modified OP2 API

To fully exploit the potentials of the emerging technology, we modify the OP2 API to use a future based model. In
new_data1=op_par_loop_save_soln("save_soln",cells,
    op_arg_dat(data_a0,...),...,
    op_arg_dat(data_an,...);
new_data2=op_par_loop_adt_calc_("adt_calc",cells,
    op_arg_dat(data_b0,...),...,
    op_arg_dat(data_bn,...);  
new_data2.get();
new_data3=op_par_loop_res_calc("res_calc",edges,
    op_arg_dat(data_c0,...),...,
    op_arg_dat(data_cn,...);  
new_data3.get();
new_data4=op_par_loop_bres_calc("bres_calc",bedges,
    op_arg_dat(data_d0,...),...,
    op_arg_dat(data_dn,...);  
new_data4.get();
new_data5=op_par_loop_update("update",cells,
    op_arg_dat(data_e0,...),...,
    op_arg_dat(data_en,...);  
new_data1.get();
new_data5.get();

Figure 10: Airfoil.cpp is changed while using async and
par(task) for loop parallelization in OP2. new_data is re-
turned from each kernel function after calling op_par_loop and
new_data.get() is used to get futures ready before the next
steps.

Figure 2, op_arg_dat creates an argument that is passed
to a kernel function through op_par_loop. The modified
OP2 API passes the argument as a future created with the
modified op_arg_dat, which uses dataflow. In using
dataflow, if an argument is a future, then the invocation
of a function will be delayed. Non-future arguments are passed
through. Figure 11 shows the schematic of a Dataflow object.
A Dataflow object encapsulates a function $F(\text{in}_1,\text{in}_2,...,\text{in}_n)$
with $n$ inputs from different data resources. As soon as the last
input argument has been received, the function $F$ is scheduled
for execution [5]. The main advantage of a data-flow based ex-
ecution is minimizing the total synchronization by scheduling
overheads and executing a function asynchronously.

Figure 12 shows the modified op_arg_dat and dat
expressed at the last line of the code invokes a function only
once it gets ready. unwrapped is a helper function in HPX,
which unwraps the futures for a function and passes along the
actual results. All these future arguments are passed to the
kernel functions through op_par_loop.

dataflow with for_each is implemented for a loop
parallelization that makes the invocation of the loop asyn-
chronous. In Figure 13, dataflow returns a future
of arg.dat representing the result of a function and
produces asynchronous execution of loops. Based on the
Dataflow performance shown in Figure 11, the invocation of
op_par_loop_adt_calc in Figure 13 will be delayed
until all of the future arguments become ready. All arguments

Figure 11: A Dataflow object encapsulates a function $F(\text{in}_1,\text{in}_2,...,\text{in}_n)$
with $n$ inputs from different data resources. As soon as the last input argument has been received, the function $F$ is scheduled for an execution [5].
passed to each kernel functions are future except the name of a function and op_set passed to a loop. for_each(par) is used here for a loop parallelization within each kernels. It should be noted that the function represented in this Section is the same as a function in Section III-A1 but asynchronous. The futures returned represent the results as a dependency tree, which represents the execution graph that is automatically created. As a result, a modified OP2 API and dataflow gives the ability of asynchronous task execution.

In this method Airfoil.cpp is changed. Figure 14 shows only op_par_loop_adt_calc and op_par_loop_res_calc from Airfoil.cpp. Each kernel function returns an output argument as a future stored in data[t], where t is a time step, and this future depends on the futures from a previous step, which is data[t-1]. We can see that the problem of manually putting new_data.get() addressed in Section III-A2 is solved here. Moreover, dataflow provides a way of interleaving execution of indirect loops and direct loops together. Interleaving execution of direct loops can be done during compile-time, however it is almost difficult to interleave indirect loops during compile-time. Using Future based techniques in HPX such as dataflow enables having indirect loop interleaving during a run-time.

OP2 source-to-source translator with Python is modified here and dataflow with for_each is automatically produced for each loop within the Airfoil application instead of #pragma omp parallel for. The experimental result is presented in detail in Section IV.

IV. Experimental Results

The experiments for this research are executed on a the test machine that has two Intel Xeon E5-2630 processors, each with 8 cores clocked at 2.4GHZ and 65GB. Hyper-threading is enabled. The OS used by the shared memory system is 32-bit Linux Mint 17.2. OpenMP linking is done through version 5.1.0 GNU compiler. The HPX version 0.9.11 [1] is used here.

Fig.15 shows the execution time of the Airfoil application using #pragma omp parallel for, for_each, async and dataflow. It is illustrated that HPX and OpenMP has by an average the same performance on 1 thread. We are however able to improve a parallel performance in using async and dataflow for more number of threads.

To evaluate the HPX performance for loop parallelization generated with OP2, we perform strong scaling and weak scaling experiments. For the speedup analysis, we use strong scaling, for which the problem size is kept the same as the number of cores are increased. Figure 16 shows the strong scaling data for the following three loop parallelization methods: #pragma omp parallel for and for_each(par) with automatically determination of chunk_size for all loops and with static chunk_size for the large loops within the Airfoil application as explained in Section III-A1. Figure 16 illustrates that for_each(par) with the static chunk_size for the large loops has better performance than automatically determining chunk_size for all loops, since for the large loops, automatically determining their grain size will affect the application’s scalability. Also, it can be seen that OpenMP still performs better than HPX in this example.

Figure 17 shows strong scaling comparison results for #pragma omp parallel for and async with for_each(par(task)) from Section III-A2. The performance is better for async with for_each(par(task)), which is the result of the asynchronous execution of loops through the use of futures.

Figure 18 shows the strong scaling comparison results for #pragma omp parallel for and dataflow. for_each(par) is used for the loop parallelization dataflow as discussed in Section III-B. Figure 18 illustrates better performance for dataflow which is due to the asynchronous task execution through the use of futures. dataflow automatically generated an (implicit) execution tree, which represents a dependency graph and allows execution of functions asynchronously. Asynchronous task execution removes unnecessary global barriers and as a result improves scalability for parallel applications. The goal of using future is to enable the computation to continue as far as possible and to eliminate global barriers when using for_each(par) and #pragma omp parallel for. As a result, using futures allows the continuation of the current computations without waiting for the computations of the previous step, if the results are not needed. Figures 17 and 18 show that removing the global barrier synchronizations improves the parallelization performance.

By considering the above results, we can see the improvement in the performance over the OP2 (initial) version. For 32 threads in Figure 17, async with for_each(par(task)) improves the scalability by about 5% and in Figure 18, dataflow improves the scalability by about 21% compared to #pragma omp parallel for. These results show
Figure 15: Comparison results of execution time between #pragma omp parallel for, for_each, async, and dataflow used for the Airfoil application.

Figure 16: Comparison results of strong scaling between #pragma omp parallel for and for_each(par) with automatically determined and static chunk_size used for the Airfoil application with up to 32 threads. HPX allows controlling a grain size while using for_each to improve scalability. It shows a better performance for for_each with the static chunk_size compared to the auto chunk_size for small loops. Hyperthreading in enabled after 16 threads.

Figure 17: Comparison results of strong scaling between #pragma omp parallel for and async with for_each(par(task)) used for the Airfoil application with up to 32 threads. It shows a better performance for async due to the asynchronous task execution. Hyperthreading in enabled after 16 threads.

Figure 18: Comparison results of strong scaling between dataflow and #pragma omp parallel for used for the Airfoil application with up to 32 threads. It illustrates a better performance for dataflow for the larger number of threads, which is due to the asynchronous task execution. dataflow automatically generated an execution tree, which represents a dependency graph and allows asynchronous execution of functions. Hyperthreading in enabled after 16 threads.
good scalability achieved by HPX and indicates that it has the potential to continue to scale on a larger number of threads.

To study the effects of communication latencies, we perform weak scaling experiments, where the problem size is increased in proportion to the increase of the number of cores. Figure 19 shows weak scaling in terms of efficiency relative to the one core case using #pragma omp parallel for, for_each(par), async with for_each(par(task)) and dataflow for a loop parallelization. dataflow with the modified OP2 API has better parallel performance, illustrating the perfect overlap of communication with computation, enabled by HPX. Also, it can be seen when the problem size is large enough, there will be enough work for all threads, hiding the communication latencies behind useful work. So for the larger problem size, more parallelism can be extracted from the application which results in better parallel efficiency.

V. CONCLUSION

The work presented in this paper shows how the HPX runtime system can be used to implement C++ application frameworks. We changed the OP2 python source-to-source translator to automatically use HPX for task parallelization generated by OP2. The Airfoil simulation written in OP2 is used to compare the HPX performance with OpenMP that is used in OP2 parallel loops by default. Airfoil had the same performance using HPX and OpenMP running on 1 thread of the various test application cases, but we were able to obtain 5% scalability improvement in using async and 21% scalability improvement in using dataflow for loop parallelization compared with OpenMP running on 32 threads.

HPX is able to control a grain size at runtime by using for_each and as a result the resource starvation is reduced as well. However using par as an execution policy introduces the global barriers at the end of the loop same as OpenMP, which inhibits having a desired scalability. It was shown that the global barrier synchronization was removed by a future based model used in for_each(par(task)), async and dataflow, which results in significantly improving a parallelism level. future allows the computation within a loop to be processed as far as possible. Also, for using dataflow, OP2 API is modified to take full advantage of all available parallel resources through HPX. It was shown that dataflow implemented in the modified OP2 API gives a capability of automatically interleaving consecutive direct and indirect loops together during a runtime. As a result, it decreases the effects of SLOW factors and enables seamless overlap of communication with communication, which helps in extracting the desired parallelism level from an application.

Acknowledgements

We would like to thank Patricia Grubel from New Mexico State University and Parsa Amini from Louisiana State University for their invaluable and helpful comments and suggestions to improve the quality of the paper. This work was supported by the NSF award 1447831.

REFERENCES


