Computer Science – Research and Development manuscript No.
(will be inserted by the editor)

Application of the ParalleX Execution Model to Stencil-based Problems

T. Heller · H. Kaiser · K. Iglberger

Received: date / Accepted: date

Abstract In the prospect of the upcoming exa-scale era with millions of execution units, the question of how to deal with this level of parallelism efficiently is of time-critical relevance. State-of-the-Art parallelization techniques such as OpenMP and MPI are not guaranteed to solve the expected problems of starvation, growing latencies, overheads, and contention. On the other hand, new parallelization paradigms promise to efficiently hide latencies and contain starvation and contention.

In this paper we analyze the performance of one novel parallelization strategy for shared and distributed memory machines. We will focus on shared memory architectures and compare the performance of the ParalleX execution model against the quasi-standard OpenMP for a standard stencil-based problem. We compare in detail the OpenMP implementation of two applications of Jacobi solvers (one based on regular grid and one based on an irregular grid structure) with the corresponding implementation of these applications using HPX (High Performance ParalleX), the first feature-complete, open-source implementation of ParalleX, and analyze the results of both implementations on a multi-socket NUMA node.

Keywords Iterative Solvers · ParalleX · Execution model · High Performance ParalleX (HPX) · OpenMP · Parallel Algorithms · Parallel Runtime Systems · Dataflow

T. Heller
Friedrich-Alexander University Erlangen-Nuremberg,
91058 Erlangen, Germany,
E-mail: thom.heller@gmail.com

H. Kaiser
Center for Computation and Technology,
Louisiana State University,
Baton Rouge, LA 70803, USA
E-mail: hkaiser@cct.lsu.edu,
http://www.cct.lsu.edu/~hkaiser

K. Iglberger
Friedrich-Alexander University Erlangen-Nuremberg,
91058 Erlangen, Germany,
E-mail: klaus.iglberger@zisc.uni-erlangen.de,
http://www.zisc.uni-erlangen.de

1 Introduction

High Performance Computing (HPC) is currently undergoing major changes, provoked by the increasing challenges of programming and managing increasingly heterogeneous multicore architectures and large scale systems. Estimates show that at the end of this decade Exaflops computing systems consisting of hundreds of millions of cores and exposing billion-way parallelism may emerge. This paper describes an experimental execution model, ParalleX, that addresses these challenges through changes in the fundamental model of parallel computation from that of the communicating sequential processes (e.g. MPI) to an innovative synthesis of concepts involving message-driven work-queue execution in the context of a global address space [1].

The focus of this paper is the description of a novel approach for the implementation of iterative solvers of linear systems of equations, based on methods such as Jacobi- or Gauss-Seidel algorithms using the HPX runtime system. HPX is the first open source implementation of the concepts of ParalleX for conventional systems (SMPs and commodity clusters). While HPX is still experimental and moving at a fast pace with a broad range of optimization possibilities, we present early performance results on a NUMA architecture. The approach presented in this paper can easily be extended to use distributed, heterogeneous systems.

Since the interplay of hardware and software environment is quickly becoming one of the dominant factors in the design of well integrated, energy efficient, large-scale systems, we also explore the implications of the ParalleX model on the organization of parallel computing on today's NUMA architectures. We present scaling and performance results of two implementations of the Jacobi Method, written based on HPX and using the well known OpenMP parallel programming paradigm.

The remainder of this paper is structured as following: Sec. 2 gives an introduction to the ParalleX execution model and its implementation in the HPX runtime system. Sec. 3 briefly introduces the Jacobi Method and gives a description of the problem that is solved by our implementations. Sec. 4 introduces the reference OpenMP implementation and Sec. 5 gives a detailed explanation of the Jacobi Method implementation with HPX. Sec. 6 summarizes the platform on which we analyzed the performance aspects. Sec. 7 provides an analysis of the performance behavior of the implementations discussed in the previous sections, before Sec. 8 concludes the paper and provides suggestions for future work.

2 The ParalleX Execution Model and its Implementation using HPX

The main objective of the ParalleX execution model [1] is to address the key challenges of efficiency, scalability, sustained performance, and power consumption of applications with respect to the limitations of conventional programming practices (e.g., OpenMP or MPI).

2.1 The ParalleX Execution Model

ParalleX tries to improve efficiency by reducing average synchronization and scheduling overhead, improve utilization through asynchrony of workflow, and employ adaptive scheduling and routing to mitigate contention (e.g., memory bank conflicts). Scalability will be increased, at least for certain classes of problems, through data directed computing using message-driven computation and lightweight synchronization mechanisms that will exploit the parallelism intrinsic to dynamic directed graphs through their meta-data. As a consequence, sustained performance will be improved both in absolute terms through extended scalability for those applications currently constrained, and in relative terms due to enhanced efficiency achieved. Finally, power reductions will be achieved by reducing extraneous calculations and data movements. The key aims of ParalleX are a) to expose new forms of program parallelism to increase the total amount of concurrent operations; b) to reduce overheads improving efficiency of operation and, in particular, to make effective use of fine grain parallelism where it should occur (this includes, where possible, the elimination of global barriers), and c) to facilitate the use of dynamic methods of resource management and task scheduling to exploit runtime information about the execution state of the application and permit continuing adaptive control for best causal operation. ParalleX is solidly rooted in the following governing principles:

- The utilization of an Active Global Address Space (AGAS) spanning the whole machine, without the assumption of cache coherence. This implies the preference of using adaptive locality management over purely static data placement strategies.
- The exposure of new forms of program parallelism, including fine grain parallelism, a fundamental paradigm shift from communicating sequential processes (CSP [2]), MPI [3] and OpenMP [4] as today's prevalent programming models.
- The preference for mechanisms, which allow to hide latencies over methods for latency avoidance.
- The preference to move work to the data over moving data to the work.
- The elimination of global barriers, replacing them with constraint-based synchronization techniques built on Local Control Objects (LCOs) to enable the efficient use of fine-grain parallelism.

2.2 The High Performance ParalleX Runtime System

High Performance ParalleX (HPX [1][5][6]) is the first open-source implementation of the ParalleX execution model. HPX is a state-of-the-art runtime system developed for conventional architectures and, currently, Windows and Linux-based systems; e.g. large Non Uniform Memory Access (NUMA) machines and clusters. Strict adherence to Standard C++11 and the utilization of the Boost C++ Libraries [7] makes HPX both portable and highly optimized. It is modular, feature-complete and designed for optimal performance. This modular framework facilitates simple compile- or runtime configuration and minimizes the runtime footprint. HPX supports both dynamically loaded modules and static prebinding at link time. The design of HPX is focused on overcoming conventional limitations such as global barriers, poor latency hiding and lack of support for fine-grain parallelism. The current implementation of HPX supports all of the key ParalleX paradigms; Parcels, PX-threads, Local Control Objects (LCOs), the Active

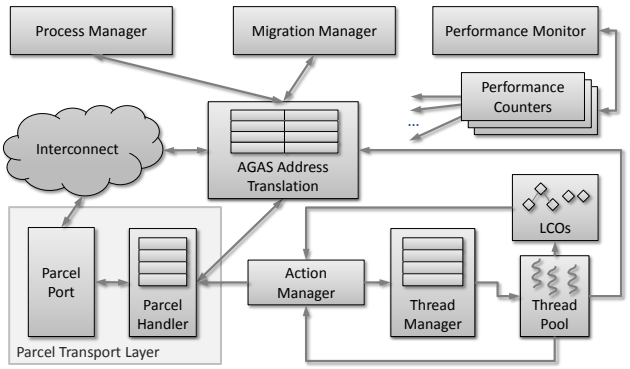

Fig. 1 Architecture of the HPX runtime system. HPX implements the supporting functionality for all of the elements needed for the ParalleX model: Parcels (parcel-port and parcel-handlers), HPX-threads (thread-manager), LCOs, AGAS, HPX-processes, performance counters and a means of integrating application specific components. An incoming parcel (delivered over the interconnect) is received by the parcel port. One or more parcel handlers are connected to a single parcel port, optionally allowing to distinguish different parts of the system as the parcel's final destination. An example for such different destinations is to have both normal cores and special hardware (such as a GPGPU) in the same locality. The main task of the parcel handler is to buffer incoming parcels for the action manager. The action manager decodes the parcel, which contains an action. An action is either a global function call or a method call on a globally addressable object. The action manager creates a PX-thread based on the encoded information.

Global Address Space (AGAS), and PX-processes (see Fig. 1).

Many HPX applications, including the linear algebra applications detailed here, utilize Local Control Objects (LCOs) to simplify parallelization and synchronization. Simple, but useful examples of LCOs are Futures [8][9] and Dataflow objects [10]. The linear algebra applications detailed in this paper make extensive use of both types of LCOs.

As shown in Fig. 2, a Dataflow object encapsulates a delayed computation. It acts as a proxy for a result initially not known, most of the time because the input arguments required for the computation of the result have not been computed yet. The Dataflow object synchronizes the computation of an output value by deferring its evaluation until all input values are available. Thus a Dataflow object is a very convenient means of constraint-based synchronization of the overall execution. We leverage the inherent meta information of the used data structures (i.e. the structure of the computational grid) by creating a structurally matching network of Dataflow objects. For this reason the execution of the dataflow network will always yield the correct result while the existing compute resources are optimally utilized as every particular calculation of any intermediate result will happen 'at its own pace'. The main advantage of such a dataflow-based execution is that it enables minimizing the total synchronization and scheduling overheads.

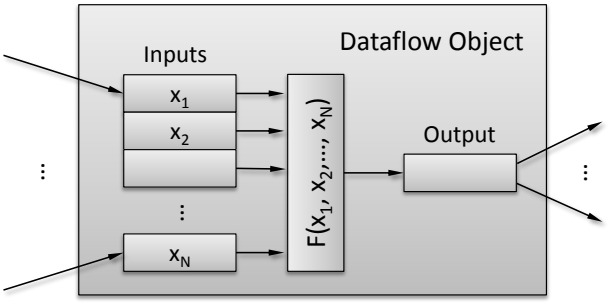

Fig. 2 Schematic of a Dataflow object. A Dataflow object encapsulates a (customizable) function $F(x_1, x_2, ..., x_N)$. This function takes N input arguments and produces an output result. The dataflow object is receiving each of the required input arguments separately from different data sources (i.e. other dataflow objects). As soon as the last input argument has been received, the function is scheduled for execution and, after the result has been calculated, it is sent to all objects connected to the output of the Dataflow object.

3 The Jacobi Method

The Jacobi Method is a numerical stationary iterative method. It is one example of stencil-based iterative methods. It is used to solve the linear system of equations given by:

$$Ax = b,\ A \in \mathbb{R}^{NxN},\ x \text{ and } b \in \mathbb{R}^N$$

The Jacobi Method is a well-known numerical algorithm with well known techniques to parallelize it with OpenMP.

The algorithm can be formulated in the following way:

$$A = D + R \text{ where}$$

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \quad R = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1N} \\ a_{21} & 0 & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & 0 \end{bmatrix},$$

$$\Rightarrow x^{(k+1)} = D^{-1}(b - Rx^{(k)})$$

The exact derivation and algorithm can be found in [11].

3.1 Problem 1: Uniform Grid

As the first problem we are solving we chose a uniform grid, which is derived by discretizing the diffusion equation for a scalar function $u(x, y)$

$$\triangle u = f$$

on a rectangular grid with Dirichlet boundary conditions. After discretizing the differential operator, we can derive the five-point stencil (without the loss of genericity, we restrict ourselves to 2D in this paper and a right-hand-side of constant zero) to be used for updating one point:

$$u_{new}(x,y) = \frac{u_{old}(x+1,y) + u_{old}(x-1,y)}{4} + \frac{u_{old}(x,y+1) + u_{old}(x,y-1)}{4}$$

Listing 1 shows an example implementation in C++ for that particular update. The type `grid<double>` is a user defined type to represent the discretized solution. It is a straightforward implementation of a linearized two-dimensional data structure.

Listing 1 Example kernel for the Jacobi Method

```
typedef pair<size_t, size_t> range_t;
void jacobi_kernel(
  vector<grid<double> > & u
, range_t x, range_t y
, size_t dst, size_t src)
{
  for(size_t j = y.first; j < y.second; ++j)
  {
    for(size_t i = x.first; i < x.second; ++i)
    {
        u[src](i,j)
         = (u[dst](i+1,j) + u[dst](i-1,j)
            + u[dst](i,j+1) + u[dst](i,j-1))
           * 0.25
    }
  }
}
```

This implementation of the kernel counts 4 double precision floating point operations (three additions and one multiplication). We assume that the cache is large enough, such that three rows of the grid are resident in the cache. According to that assumption, three memory transfers have to be done for every iteration. This implies that this specific algorithm is memory bound. In this example implementation of the kernel we don't care about Low-Level optimizations. Those optimizations are heavily discussed in the literature [12], and have no effect on the outcome of this comparison. Additionally, no convergence test is performed in order to have a fixed number of iterations for both implementations.

3.2 Problem 2: Non-Uniform Grid

The second problem is chosen to show how different workloads on an element update influences our target implementations. We choose a matrix obtained from a Finite Element Method. The matrix is obtained from a structural problem discretizing a gas reservoir with tetrahedral Finite Elements [13][14]. The matrix is saved in a compressed row storage format to efficiently traverse it in the Jacobi Method Kernel. For simplicity an implementation of the sparse matrix data structure and the concrete kernel implementation is not shown. As Problem 1, this problem is memory bound. The resulting stencil is dynamic, and dependent on how many entries the row of the element to update has. The chose n matrix has a minimum of one entry per row, and a maximum of 228 entries. The mean are 23.69 entries, with at least one element on the diagonal. We assume that the diagonal element of the chosen matrix, and the element of the source grid are cache resident. Based on these numbers, the mean memory transfers for this problem are 47.38 (2 for the destination grid and the right hand side, 22.69 for the matrix elements and 22.69 for the source grid). These numbers are the baseline for the performance discussion in Sec. 7.

4 OpenMP Implementation

OpenMP is the dominant shared-memory programming standard today. Developed since 1997 as a joint effort of compiler vendors, OpenMP has been used even prior to the multicore era as a platform independent parallelization tool.

Listing 2 shows the OpenMP-parallel implementation of the Jacobi solver for Problem 1 (Sec. 3.1). For the sake of simplicity, only the implementation of Problem 1 is shown, not Problem 2 (Sec. 3.2). Initially, the two grids are initialized with `1` according to the Dirichlet boundary conditions. We iterate over the grid in a block-wise fashion in order to use the cache efficiently. In every inner loop, the function outlined in Listing 1 is executed on each single block.

Listing 2 OpenMP - Jacobi Method

```
vector< grid<double> >
  u( 2, grid<double>(N_x, N_y, 1) );
size_t src = 0;
size_t dst = 1;
for(size_t iter = 0; iter < max_iter; ++iter){
#pragma omp parallel for shared(u) schedule(
    static)
  for(size_t y=1; y < N_y-1; y+=block_size)
  {
    for(size_t x=1; x < N_x-1; x+=block_size)
    {
      jacobi_kernel(
         u
       , range_t(x, min(x+block_size, N_x))
       , range_t(y, min(y+block_size, N_y))
       , src, dst);
    }
  }
  swap(src, dst);
}
```

This algorithm uses standard C++ facilities almost exclusively and hints the compiler to create a parallel section with `#pragma omp parallel for`. Due to the nature of the Jacobi method, this algorithm does not contain any data races. It is worth noting that with the directive `#pragma omp parallel for` an implicit global barrier is introduced. This barrier has the effect that every thread completed the calculation and the grids can be swapped to initiate the next iteration. However, as the workload of the calculation is equally distributed among each thread, this barrier should have little to no effect.

Since the Jacobi method is a memory-bound algorithm, performance of this implementation is expected to scale well until the maximum memory bandwidth is exhausted.

In case of applications that are memory-bound, locality and contention problems can occur on NUMA systems if no special care is taken for the memory initialization. Memory-bound code must be designed to employ proper page placement, for instance by first touch [12]. In order to accommodate both use-cases, we use an interleaved memory policy. That means that pages will be allocated using round robin on nodes. This decreases the overall performance for Problem 1, but is the most practical way of placing memory for Problem 2. Additionally, thread migration was prevented by an explicit pinning of all active threads via the Likwid tool suite [15]. Without applying these techniques, OpenMP would fail to scale properly on NUMA architectures.

5 HPX Implementation

The HPX implementation of Problem 1 (Sec. 3.1) eliminates the implicit global barriers introduced by parallelizing the iteration loops using OpenMP. This is achieved by using LCOs introduced by ParalleX (see Sec. 2.1) and implemented in HPX (see Sec. 2.2). For the sake of simplicity, only the implementation of Problem 1 is shown, not Problem 2 (Sec. 3.2).

In particular the Dataflow LCO is used to ensure a fluid flow of computation. The Jacobi Method can then be formulated with the help of Dataflow by explicitly specifying the dependencies. In the case of the algorithm outlined above (See Listing 1), the constraints are that the elements of the old grid had been computed so that we don't create a data race. In the case of the OpenMP Listing, this has been ensured by the implicit global barrier (See Listing 2). The HPX implementations solves this by explicitly calculating the block-wise dependencies.

Listing 3 HPX - Calculating dependencies

```
1 typedef hpx::lcos::dataflow<void> dataflow_t;
2
3 vector<dataflow_t> get_deps(
4   grid<dataflow_t> const & deps
5 , size_t x, size_t y)
6 {
7   vector<dataflow_t> r;
8   r.push_back(deps(x,y));
9   if(x>0)          r.push_back(deps(x-1,y));
10  if(x+1<deps.N_x) r.push_back(deps(x+1,y));
11  if(y>0)          r.push_back(deps(x,y-1));
12  if(y+1<deps.N_y) r.push_back(deps(x,y+1));
13  return r;
14 }
```

In Listing 3 the block-wise dependency calculation is shown: The current item has to be calculated in order to be able to continue (Line 8); The left item, iff we are not at the left boundary (Line 9); The right item, iff we are not at the right boundary (Line 10); The top item, iff we are not at the top boundary (Line 11); The bottom item, iff we are not at the bottom boundary (Line 12); The dependencies can be trivially derived by looking at the stencil.

Listing 4 HPX - Jacobi Method

```
1  typedef dataflow<void> dataflow_t;
2  typedef
3    vector<grid<double> > vector_t;
4  hpx::components::dataflow_object<vector_t>
5    u = new_<vector_t>(hpx::find_here(),
6        2, grid<double>(N_x, N_y, 1));
7  grid<dataflow_t>
8    deps(N_x/block_size, N_y/block_size);
9  size_t dst = 0;
10 size_t src = 1;
11 for(size_t iter = 0; iter < max_iter; ++iter)
12 {
13   for(size_t y = 1, y_block = 0;
14       y < N_y-1; y += block_size, ++y_block)
15   {
16     for(size_t x = 1, x_block = 0;
17         x < N_x-1; x += block_size, ++x_block)
18     {
19       vector<dataflow_t>
20         iteration_deps = get_deps(
21           deps, x_block, y_block);
22       deps(x_block, y_block)
23         = u.apply(
24           hpx::bind(jacobi_kernel,
25             hpx::placeholder::_1
26           , range_t(x, min(x+block_size, N_x))
27           , range_t(y, min(y+block_size, N_y))
28           , dst, src
29           )
30         , iteration_deps
31       );
32     }
33   }
34   swap(dst, src);
35 }
36 hpx::wait(deps);
```

Listing 4 shows the full implementation of the Jacobi Method with HPX. It is still 100% C++ Standard com-

pliant, but implements a complete novel approach to parallelism. Instead of hinting the compiler to parallelize the loop, we have the parallelism hidden inside the Dataflow LCO. This Dataflow LCO is responsible for activating a new thread once all dependencies have been met (see Listing 3). On Line 6 a new object is created and registered with AGAS (see Sec. 2.2) and wrapped behind a special datatype. The iterations look exactly like in Listing 2 but are now used to create the Dataflow tree. At first, we need to determine the dependencies of the current iteration (Line 21). The call to `u.apply` (Line 23) creates the dataflow object. With `hpx::bind` (Line 24) a function object is created that will be executed once the passed dependencies (Line 30) are met. This complies to the "moving work to data" paradigm imposed by the ParalleX execution model (see Section 2). For what it matters, whether the object lives on the current system, or on a remote location is completely irrelevant for this code to work.

It is important to note that after the loops are finished, the system already started working in the background. What we achieved here is to remove the implicit global barrier imposed by OpenMP with constraint based LCOs, which control, with the help of the runtime system, all the introduced parallelism.

As the OpenMP implementation, the HPX implementation is memory bound. Except for pinning the operating system threads to the according cores, no other precautions to accommodate NUMA architectures have been done. Due to the aggressive thread stealing of the HPX thread-manager (see Sec. 2.2) an optimal NUMA memory placement strategy is yet to be found. Although, this thread stealing also implies the possibility to hide latencies induced by inter-node memory traffic, it cannot compensate the intrinsic bandwidth limits of current computing platforms.

6 Benchmark Platform

In order to evaluate the performance properties of the Jacobi Method implementations, we use a multi-socket AMD Istanbul platform. The nominal specifications of this machine are:

- 8 Sockets, one AMD Opteron 8431 each socket (48 Cores in total) 6 Cores per socket, 2.4 GHz Base speed; per Core: L1 Cache Size: 128 KB, L2 Cache Size: 512 KB, shared: L3 Cache Size: 6144 KB
- 2 GB RAM per Core (96 GB RAM in total)
- Maximum 8.5 GB/s Memory Bandwidth per socket.
- Theoretical double precision peak performance: 230 GFLOPS/s

Apart from the nominal numbers, we run the STREAM benchmark [16] to measure the maximum memory bandwidth. Fig. 3 shows the result of the STREAM benchmark Copy test. Due to memory bound nature of the discussed algorithm (see Sec. 3), we use those experimental results as the basis of our considerations. Deriving from theses numbers, the maximum achievable speedup is 4.88 and scaling beyond four cores cannot be expected. Additionally, the implications of non-optimal memory placement can be observed by considering the difference between the theoretical bandwidth and the measured. By placing the allocated memory pages in a round-robin fashion (interleaved) we get a good estimate on how much performance can be expected in our benchmarks.

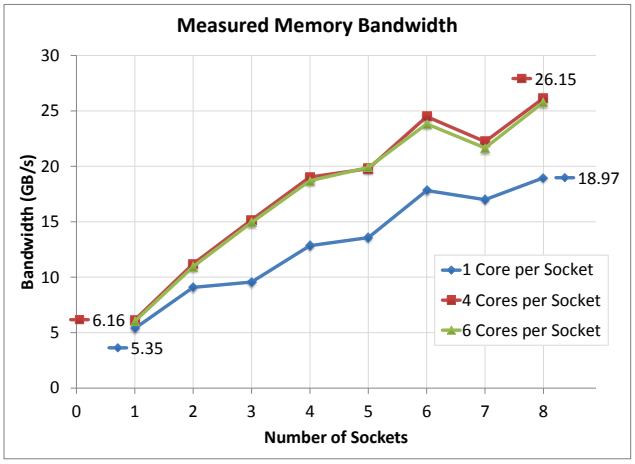

Fig. 3 Measured memory bandwidth on the benchmark platform. The shown data allows to derive that the best possible speedup when scaling from 1 to 48 cores on this particular system is 4.88 ($\frac{26.15}{5.35}$) Further, this figure shows that increasing the number of utilized cores per socket beyond four does not yield any performance benefit (the scaling numbers for six cores per socket are smaller than those for four cores).

7 Results

We benchmark our implementations using the example problems introduced in Sec. 3. For the uniform problem (Sec. 3.1), we performed weak scaling and strong scaling tests. For weak scaling, the number of points on the y-axis were kept constant at 100000, and for every core 1000 grid points on the x-axis were added. For strong scaling, we chose a problem size of 10000x100000 (`N_x = 10000` and `N_y = 100000`). The `block_size` has been chosen to be 1000 such that one tile calculation fits easily into the level-2 cache of our platform. Additionally, the problem size has been chosen big enough such that enough work for all processors can be assumed. The algorithm has been run for 100 iterations. The Jacobi Solver for the irregular grid discussed in Sec. 3.2 was run for 1000 iterations and a `block_size` of 50000 has

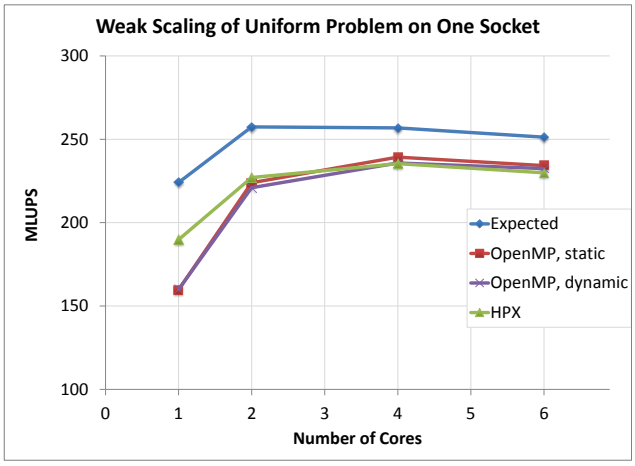

Fig. 4 Weak scaling of the uniform problem on one socket. The measured results for the OpenMP and HPX applications are compared with the measured maximum bandwidth (as derived from Fig. 3). Both, the OpenMP and HPX applications have similar qualitative scaling behavior. All three implementations show good weak scaling.

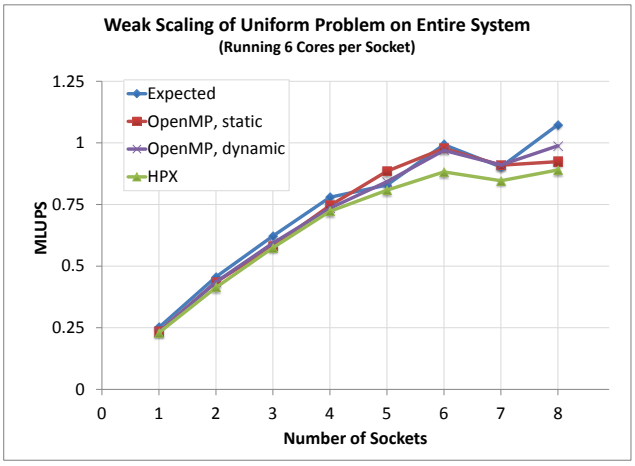

Fig. 5 Weak scaling of the uniform problem on the entire system. The measured results for the OpenMP and HPX applications are compared with the measured maximum bandwidth for the whole system (as derived from Fig. 3). These numbers are qualitatively consistent with the results presented in Fig. 4.

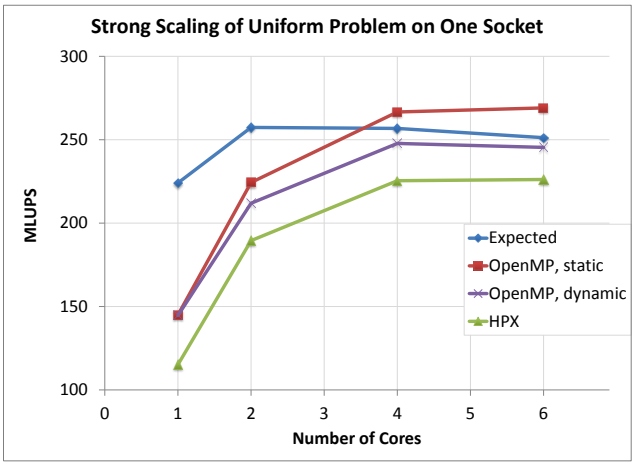

Fig. 6 Strong scaling of the uniform problem on the entire system. The measured results for the OpenMP and HPX applications are compared with the measured maximum bandwidth for the whole system (as derived from Fig. 3). These numbers are qualitatively consistent with the results presented in Fig. 6.

been chosen such that the level-2 cache is employed appropriately.

The performance is measured in Mega Lattice Site Updates per second (MLUPS). One lattice site is one element of the grid. Based on the performance characteristics of the benchmark platform (Sec. 6), we can calculate the maximal achievable performance. The update of one lattice in the uniform grid needs three memory transfers, thus 24 bytes are transferred. In the case of the non-uniform grid, we assume the mean number of memory transfers of 47.38, which means that 379.04 bytes are transferred for one lattice update. Dividing the measured peak memory bandwidth by those numbers, leads to an expected maximum performance (shown as blue lines in all figures). For the OpenMP implementations we measure the achieved performance with static and dynamic work scheduling.

All of our benchmarks shows the same weak scaling behavior for the uniform testcase. It can be observed that the HPX implementation is able to perform and scale as good as the OpenMP implementation (Fig. 4).

Weak scaling on the entire system (Fig. 5) is similar to scaling on one core. When utilizing the complete machine, the presented implementations are able to reach the expected performance almost everywhere. HPX performance starts to decline after using more than 5 sockets. This is due to the currently non-optimized NUMA placement (a maximum of 9% slower).

As expected for the uniform workload, OpenMP is able to perfectly use the resources of the system, while HPX is around 15% slower (Fig. 6). This is due to runtime overheads of the HPX runtime system, which cannot be amortized by such a highly regular workload. The scaling behavior however is consistent for both implementations.

Fig. 7 shows the performance of both implementations for the entire system. For both runs, interleaved memory placement is used. It can be seen that the OpenMP implementation is able to almost reach the expected performance. The HPX implementation show the same scaling behavior. By using more parallel compute resources, the HPX implementation is able to reach the OpenMP performance. Changing the scheduling of the OpenMP work chunks, only influences the results minimally.

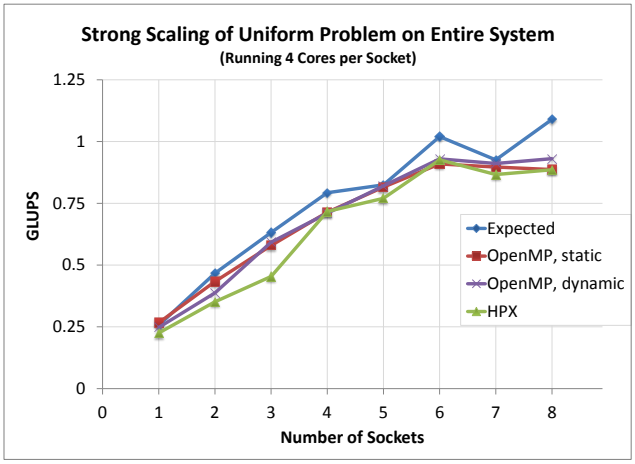

Fig. 7 Strong scaling of the uniform problem on the entire system. The measured results for the OpenMP and HPX applications are compared with the measured maximum bandwidth for the whole system (as derived from Fig. 3). These numbers are qualitatively consistent with the results presented in Fig. 6.

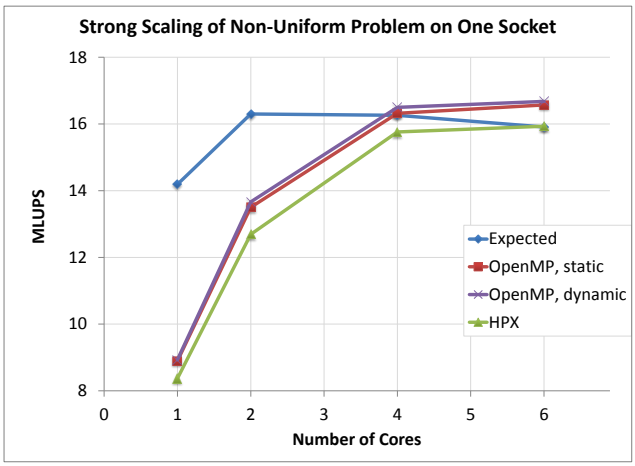

Fig. 8 Strong scaling of the non-uniform problem on one socket. Both, the OpenMP and HPX applications have similar qualitative scaling behavior, while the HPX application consistently shows slightly slower runtimes. However, the difference is much smaller than in the uniform case.

The performance behavior of the non-uniform problem on one socket (Fig. 8) shows similar properties as the regular problem. Due to the workload imbalance, the dataflow-based parallelization technique of HPX is able to gain ground. It is now only 6% slower than the OpenMP one.

In Fig. 9 the benefits of the dataflow-based implementation with HPX can be seen very clearly. The more parallel resources are added to the system, the better the HPX implementation is able to perform. It can be observed that a maximum of 18% performance gain (24 cores) over the OpenMP implementation with static scheduling can be reached. This can be explained by

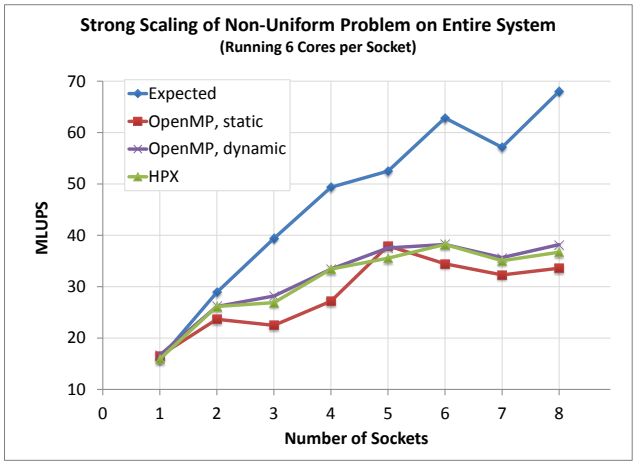

Fig. 9 Strong scaling of the non-uniform problem on the entire system. This figure shows, that the HPX application performs better than the OpenMP code. The advantage increases with an increasing number of parallel resources utilized for the runs.

the fact that every element update has different timing behaviors due to the irregular mesh. While HPX is able to dynamically schedule work items, the implicit global barrier of OpenMP needs to wait for all blocks to finish. It can be seen that by using 5 Sockets (30 cores), the workload is balanced, and OpenMP is able to reach a higher performance. By changing the work scheduling algorithm of OpenMP to be dynamic, both HPX and OpenMP show almost the same performance, with slightly better results for OpenMP.

8 Conclusion

In this paper we compared the performance and scaling characteristics of two different applications of a Jacobi solver for a linear system of equations. One application is using a very regular, uniform grid, whereas the other solves the linear system of equations on a highly irregular grid. Both applications have been implemented using OpenMP and HPX. We present results confirming that the data driven, task-queue based, fine grain parallelism and the constraint based synchronization methods as implemented in HPX are highly beneficial if the properties inherent to the tested algorithm cause load imbalances in terms of execution time or data distribution. Further, the implementation of a Jacobi solver based on OpenMP exposes almost perfect performance and scaling characteristics for a highly regular use case. However, if the algorithm causes more irregular resource utilization, the performance and scaling capabilities of the OpenMP solution degrade quickly, unless the scheduling of work chunks is changed. At the same time we see almost no degradation in terms of performance or scaling for an equivalent HPX appli-

cation, allowing to outperform and outscale the static OpenMP based application for the analyzed, highly irregular usage modes. While OpenMP and HPX are on the same performance level when comparing the dynamic work scheduling algorithms where HPX is able to provide a uniform solution and has a lot of potential optimization possibilities.

While this paper didn't discuss distributed memory systems it should be noted that formulating an algorithm with HPX that is able to run on clusters is easy to implement. The exchange of ghost zones can be formulated in terms of Data flow and the communication will be automatically overlapped with computation. Such an approach will lead to highly complex implementations if implemented with conventional methods such as MPI and OpenMP.

9 Acknowledgments

We thank Matthew Anderson for helpful discussions and for preparing the irregular grids we used. We thank Georg Hager for optimizing the OpenMP usage and for suggesting the applied performance model. We acknowledge the support from the Center for Computation and Technology (CCT) at Louisiana State University (LSU) and from NSF grants (1029161, 1117470) to LSU.

References

1. Kaiser, H., Brodowicz, M., Sterling, T.: ParalleX: An advanced parallel execution model for scaling-impaired applications. In: Parallel Processing Workshops, pp. 394–401. IEEE Computer Society, Los Alamitos, CA, USA (2009). DOI http://doi.ieeecomputersociety.org/10.1109/ICPPW.2009.14
2. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31**(3), 560–599 (1984)
3. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.2. High Performance Computing Center Stuttgart (HLRS), Stuttgart, Germany (2009)
4. Dagum, L., Menon, R.: OpenMP: An industry-standard api for shared-memory programming. IEEE Computational Science and Engineering **5**(1), 46–55 (1998). DOI http://doi.ieeecomputersociety.org/10.1109/99.660313
5. STE||AR Group: Systems Technologies, Emerging Parallelism, and Algorithms Reseach (2011). URL http://stellar.cct.lsu.edu
6. Anderson, M., Brodowicz, M., Kaiser, H., Sterling, T.L.: An Application Driven Analysis of the ParalleX Execution Model. CoRR **abs/1109.5201** (2011). Http://arxiv.org/abs/1109.5201
7. Boost: a collection of free peer-reviewed portable C++ source libraries (2011). URL http://www.boost.org/. http://www.boost.org/
8. Baker, H.C., Hewitt, C.: The incremental garbage collection of processes. In: SIGART Bull., pp. 55–59. ACM, New York, NY, USA (1977). DOI http://doi.acm.org/10.1145/872736.806932. URL http://doi.acm.org/10.1145/872736.806932
9. Friedman, D.P., Wise, D.S.: Cons should not evaluate its arguments. In: ICALP, pp. 257–284 (1976)
10. Papadopoulos, G., Culler, D.: Monsoon: An explicit token-store architecture. In: 17th International Symposium on Computer Architecture, no. 18(2) in ACM SIGARCH Computer Architecture News, pp. 82–91. ACM Digital Library, Seattle, Washington, May 28–31 (1990)
11. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., der Vorst, H.V.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition. SIAM, Philadelphia, PA (1994)
12. Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers, 1st edn. CRC Press, Inc., Boca Raton, FL, USA (2010)
13. Janna, C., Ferronato, M.: Janna/Serena sparse matrix (2011). URL http://www.cise.ufl.edu/research/sparse/matrices/Janna/Serena.html
14. Ferronato, M., Gambolati, G., Janna, C., Teatini, P.: Geomechanical issues of anthropogenic co2 sequestration in exploited gas fields. Energy Conversion and Management **51**(10), 1918 – 1928 (2010). DOI 10.1016/j.enconman.2010.02.024
15. Triebig, J.: Likwid: Linux tools to support programmers in developing high performance multi-threaded programs. URL http://code.google.com/p/likwid/
16. McCalpin, J.D.: STREAM: Sustainable memory bandwidth in high performance computers. A continually updated technical report, University of Virginia, Charlottesville, VA (1991–2007). URL http://www.cs.virginia.edu/stream/