OpenMP for next generation heterogeneous clusters

Jens Breitbart
Research Group Programming Languages / Methodologies,
Universität Kassel,
jbreitbart@uni-kassel.de

Abstract

The last years have seen great diversity in new hardware with e. g. GPUs providing multiple times the processing power of CPUs. Programming GPUs or clusters of GPUs however is still complicated and time consuming. In this paper we present extensions to OpenMP that allow one program to scale from a single multi-core CPU to a many-core cluster (e. g. a GPU cluster). We extend OpenMP with a new scheduling clause to enable developers to specify automatic tiling and library functions to access the tile size or the number of the currently calculated tile. We furthermore demonstrate that the intra-tile parallelization can be created automatically based on the inter-tile parallelization and thereby allows for scalability to shared memory many-core architectures. To be able to use OpenMP on distributed memory systems we propose a PGAS-like memory level called world memory. World memory does not only allow data to be shared among multiple processes, but also allows for fine-grained synchronization of processes. World memory has two states: initialized and uninitialized. A process reading from uninitialized memory will be suspended, until another process writes to that memory and thereby initializes it. This concept requires oversaturating the available hardware with processes.

1 Introduction

Current computer systems show a trend towards heterogeneity, as for example accelerators offer better performance compared to generic CPUs in common scenarios. However, the gain in processing power is currently not joined by a gain in increased software development productivity, as the programming systems for accelerators are still at a rather early development stage. For example, OpenCL [4] is the first programming system enabling cross-platform compatibility between CPUs and different accelerators like GPUs and FPGAs. Developing code with OpenCL requires optimizing for the target hardware at a rather high level. To effectively use NVIDIAs GPUs [2] it is essential to utilize so called local memory, whereas e. g. IBMs OpenCL [5] implementation highly disadvises for using this memory type when compiling for PowerPC. The success of OpenCL is yet undetermined, even though it is supported by almost all major hardware vendors. Furthermore it is still questionable if OpenCL will allow for high productivity and ease of use.

Despite all problems, the high performance of accelerators drives them to be used not only in standalone PCs, but clusters as well. In clusters the problems of programming accelerators is joined by the complexity of programming distributed memory systems. In case of using GPUs as accelerators the resulting memory hierarchy is rather complex, as each GPU has a three-ary memory hierarchy all by itself. When MPI and OpenCL is used to program such a cluster, the data flow between the nodes and within GPU memory system must be managed manually, which by itself is time consuming and error-prone even though the data flow may be rather obvious.

In the last years, research on languages providing a partitioned global address space (PGAS) for distributed memory systems has resulted in a number of new languages aimed at allowing high productivity for programming distributed memory systems. Some of these languages are: Chapel, Unified Parallel C and X10. Their accelerator support is at best at an early development stage and their usability is therefore yet undetermined.

In this paper we propose extensions to OpenMP 3.0 [1] that allow scalability to hundreds of shared memory cores, as they are used in GPUs, and a PGAS like memory model to allow easy programing and scalability on distributed memory architectures without loosing the usability of OpenMP. The syntax shown in the upcoming sections is preliminary, but we expect the underling concepts to be useful.

We rely on tiling to allow scalability to a high num-

ber of shared memory cores. Tiling is a well known technique to increase data locality and to reduce possible synchronization [8]. Furthermore tiling is one of the foundations for the high performance made available by GPUs, as they normally work on tiles stored in a fast on-chip memory. To allow for high productivity, we describe tiling in an almost transparent way, so the programming system can choose the size of tiles or even decide not to apply tiling at all. We focus on a scenario in which the data dependencies between the tiles (*inter-tile*) are identical to those within the tiles (*intra-tile*) and present a way that allows for automatic parallelization of the intra-tile execution. The automatic parallelization is based on the inter-tile parallelization and allows for automatic usage of possible on-chip memory.

To support programmability and scalability on distributed memory systems, we suggest a PGAS-like memory model, for which memory has two states: uninitialized and initialized. We call this memory type *world memory*. World memory is unitialized when no process has written to it. In case a process reads from uninitialized world memory, the process is suspended until another process writes to it. Keeping processors busy requires that the distributed memory nodes must be oversatured with processes. The world memory concept allows for fine grained synchronization and reduces the need of using the classic OpenMP synchronization primitives, as they may not scale for large distributed memory systems. The world memory joined with tiling and the automatic intra-tile parallelization allows for one program to scale from a single multi-core CPU to e. g. a GPU cluster.

Besides allowing scalability, the major design goal was to obtain the usability of OpenMP and to continue to allow a clear global view at the implemented algorithm without the need to care about low level details like manually distributing the available work.

The paper is organized as follows. First, Sect. 2 gives a brief overview of the diversity of the currently available hardware and Sect. 3 describes the well known Gauss-Seidel stencil, which we use as an example throughout the following sections. The next two sections describe our extensions. At first we describe the new shared memory extensions in Sect. 4, which are joined by the distributed memory extensions in Sect. 5. Section 6 summarizes the paper.

2 Hardware overview

Current general purpose CPUs provide a decent performance for basically every workload. Programming such systems is known to be rather easy, even though some optimizations may still be complicated in most programming systems e. g. vectorizing non-trivial code or increasing cache utilization of a specific function. However, it is questionable how far the current design with multiple cores sharing levels of coherent caches will scale.

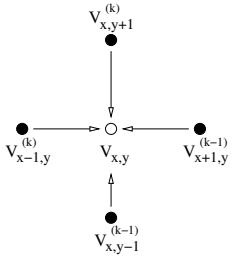

Figure 1: Gauß-Seidel data dependency

GPUs built by AMD and NVIDIA are probably the most common accelerators and provide performance that is multiple times that of CPUs. However, achieving high practical performance is only possible for highly parallel applications with predictable memory accesses. In contrast to CPUs, GPUs consist of hundreds of small cores, for which one provides far less performance than a single CPU core. The GPU cores are oversaturated with threads, so a scheduler may hide memory access latency for off-chip memory. In currently available GPUs the off-chip memory is not cached. NVIDIA has announced its upcoming Fermi architecture, which provides a two-ary cache hierarchy with one cache being shared by all cores. However, the Fermi architecture and all current architectures still expose low level on-chip scratch-pad memory that must be used to reduce accesses to the slow off-chip memory. The scratch-pad memory is shared by a closely coupled set of threads called a threadblock, for which subgroups of threads are executed in SIMD fashion. On AMD hardware the instruction executed in a SIMD fashion is not a single instruction, but a VLIW instruction that exposes instruction level parallelismif possible. The programming systems currently used to program GPUs are a tight resemblance of the hardware layout. Developers must manually apply tiling to their problem and assign such a tile to the closely coupled block of threads. Memory transfers from off-chip memory to the fast scratch-pad memory must be done explicitly. The hardware handles the SIMD execution, however branching can decrease performance.

3 Gauß-Seidel stencil

The calculations of the Gauß-Seidel stencil are being applied on a two dimensional data matrix V with the borders having fixed values. The non-border element with the coordinates (x, y) in the kth iteration is calculated by

$$V^k_{x,y} = \frac{V^k_{x-1,y} + V^{k-1}_{x+1,y} + V^k_{x,y-1} + V^{k-1}_{x,y+1}}{4}$$

Algorithm 1 Gauß-Seidel (shared memory)

```
1  volatile int *counters;
2  #pragma omp parallel for schedule(blocked)
3  for (int x=0; x<x_size-2; ++x) {
4    #pragma omp single
5    {
6      counters = new int[omp_num_blocks()+1];
7      //initialize all counters with 0
8    }
9
10   int x_block = omp_block_num();
11   #pragma omp for schedule(blocked)
12   for (int y=0; y<y_size-2; ++y) {
13     if (x==0) counters[0]=omp_total_size();
14     while (y>=counters[x_block]) {}
15
16     #pragma omp block
17     V[y+1][x+1] = (V[y+1][x] + V[y+1][x+2] +
           V[y][x+1] + V[y+2][x+1])/4;
18
19     #pragma omp atomic
20     counters[x_block+1] += omp_block_size();
21   }
22 }
```

This calculation is repeated for a fixed number of steps or until it converges. Here we assume a fixed number of steps.

Figure 1 shows a visualization of the data dependencies of the Gauß-Seidel stencil. It is important to notice that the upper and the left values are from the kth step, whereas the right and bottom value are from the $(k-1)$th step. A parallelization can be done using the wavefront pattern [6]. In a wavefront the data matrix is divided into diagonals and elements of a diagonal are calculated in parallel. Additionally to the parallelism made available by the wavefront pattern, step $k+1$ can be started before step k is completed, since after the first two diagonals of step k are calculated, all data needed for the first diagonal of step $k+1$ is available. Both concepts are used in the following two sections.

4 Shared memory extensions

In this section we introduce our OpenMP extensions for shared memory architectures and explain how they can be mapped to both multi-core CPUs and GPUs. Algorithm 1 shows an implementation of the Gauß-Seidel stencil with the extensions.

We implemented the wavefront pattern by dividing the data matrix in columns and calculate these columns in parallel. Column 0 can be calculated from the top to the bottom, whereas column $n+1$ can only be calculated as deep as column n has been calculated. In case column $n+1$ is already calculated as deep as n, the thread calculating $n+1$ must wait until the thread calculating column n has calculated more elements. We use one counter variable per column to identify how deep a column has already been calculated. The counter variables are shared by all threads, and are updated with atomic operations. The current version of OpenMP does not allow reading from a shared variable without synchronization, even if it is updated atomically. Most current hardware however only requires a fence / flush to read an atomically updated variable, so we rely on this behavior as well.

In our algorithm we use tiling to create tiles of the data matrix `V`. Tiling `V` does not only increase the cache utilization in case `V` is too large to be kept in cache, but also reduces the number of atomic updates of the counters. Instead of updating the counter every time a new value of `V` has been calculated, tiling allows us to only update the counter once per tile.

Automatic tiling The major change compared to the existing form of OpenMP is the introduction of automatic tiling, which is also known as blocking. We introduce a new scheduling clause called `blocked` to let developers annotated the loops for which tiling should be applied. We also allow nesting of these annotations. In Alg. 1 we annotated the loops in lines 3 and 12, so the compiler can apply tiling to these loops. Loop tiling divides the loop iteration space into tiles and transforms the loops to iterate over the tiles. For example, when loop tiling is applied to a single loop, the loop is split into two loops: an outer loop which loops over tiles and an inner loop which loops over the elements of a tile. In our variant of tiling, the loop is not only split into two loops, but the inner loop is also moved in front of what we call the *instruction block*.

The instruction block should contain only the code that must be executed in every loop iteration. We expect this in most cases to be the calculation of the result and no synchronization. It is identified by `#pragma omp block` and there may only be one instruction block in a tiled loop. In our example only line 17 (Alg. 1) is the instruction block, so all inner tiling loops will be moved in front of this line. When multiple loops are defined as `schedule(blocked)`, the loop order in front of the block is identical to the one of the original loops.

Tiling cannot be applied to our example without having access to the number of tiles created, as one counter per tile must be used. Furthermore, we need to have access to the size of a tile to be able to update the counters once per tile. We provide four library functions giving direct access to the tiles: `omp_num_blocks()` returns the number of tiles a loop is split into, `omp_block_num()` returns the number of the tile currently calculated by the calling thread, `omp_block_size()` returns the size of the tile and `omp_total_size()` returns the end value of the tiled loop. The functions are bound to the tiled

loop they are directly part of, meaning in our example `omp_total_size()` (Alg. 1, line 13) is bound to the second for-loop and returns `y_size-2`. Up till now `omp_total_size()` may look superfluous; however it is required when mapping this code to the GPU, as shown next.

Automatic intra-tile parallelization Tiling allows for good cache utilization on CPUs, but is also a requirement to map our code to hardware relying on tiling, as for example GPUs. However in contrast to CPUs, GPUs require the parallel execution within the tiles to utilize all available cores. We discuss the scenario, in which the intra-tile data dependencies are identical to the inter-tile ones, as we expect this to be true in most cases. However, one could explicitly use OpenMP pragmas for synchronization within the instruction block to implement a different scheme. We first outline the technique for automatic intra-tile parallelization in general and then apply it to the Gauß-Seidel example.

We define three levels of locality: *global* refers to data that is being shared by all tiles and OpenMP pragmas that effects all threads, *tile-local* refers to data that is local to a tile or synchronization effecting only the threads calculating the same tile and *thread-local* specifies data that is local to one thread. All pragmas used till now are global, as well as all variables defined as OpenMP-`shared`. All OpenMP-`private` variables are thread-local. Our algorithm will automatically copy global variables and transform global pragmas to tile-local counterparts for parallel execution within a tile.

Our algorithm requires to divide the tiled loops into three parts. The *pre-block*, which starts at the beginning of the loop and stops at the instruction-block or the start of a nested tiled loop. The *post-block* starts after the instruction-block or at the end of a nested tiled loop and continuous till the end of the loop. In our example the pre-block of the outer loop starts at line 4 and end at line 10, whereas the post-block is empty.

To allow parallel execution within a tile, the inner tiling loops will no longer only execute the instruction block, but also modified copies of their original pre- and post-blocks. All global variables used within an inner pre- or post-block are copied to tile-local counterparts prior to the start of the pre-block and all accesses in the inner pre- and post-blocks are transformed to access the tile-local variables. In the Gauß-Seidel example, `counters` is copied and replaced by a tile-local copy. The copy of `counters` can be stored in fast on-chip memory. Accesses to the loop indices of the tiled loop in the pre- and post-block are changed accordingly. OpenMP pragmas in the inner pre- and post-block are transformed into tile-local counterparts, meaning that e. g. a `single` is only performed by one of the threads calculating a tile. In our example the `single` and `atomic` pragmas are copied to tile-local version. The same holds true, for the OpenMP library functions, which now return tile-local results, e. g. `omp_num_blocks` returns the size of the currently calculated tile. All thread-local variables stay thread-local. No additional changes to the instruction block are being applied. The shown transformations expect the same synchronization functionality at both tile and global level, which is true for current GPUs. OpenCL (or CUDA) does not allow synchronization between tiles, but previous research [7] has shown that this is not a hardware limitation.

Supporting the GPU memory system is essential to achieve high performance. We suggest an additional OpenMP parameter to let developers specify, which data should be stored in on-chip memory. In our example elements of `V` should be stored in the on-chip memory; however we do not only need the tiles, but a halo with the width of one element as well. A way to specify this in an OpenMP parameter would be `block_local(V, (x,y), 1)`. This parameter should be added to line 11 Alg. 1. The parameter describes that the Variable `V` can be tiled and a tile should be stored in the on-chip memory. The tile of data is defined by variables `x` and `y` and the halo has a size of 1. This kind of specification could also be used on current CPUs, e. g. by prefetching the specified data into cache.

We have not discussed the issue that CPU and GPU currently do not share the same memory space, however Lee et al. have already shown in [3] that the existing OpenMP data-sharing attribute clauses can be used to transfer all needed data to GPU memory.

The code that would be automatically generated from our OpenMP extensions should mostly be OpenCL compatible, so we expect most OpenCL compatible hardware to support our suggested extensions.

5 Distributed memory extensions

To enable OpenMP for distributed memory we utilize a PGAS-like memory model and an additional level of parallelism. We call the set of processes executing the program the world. The new level of parallelism is called world parallelism and the memory accessible by all processes is called world memory. In world parallelism all synchronization primitives known from OpenMP are available and work at world level, meaning an OpenMP `single` is executed by only one process. It is possible to nest "normal" shared memory parallelism in world parallelism. A `single` construct in a shared memory parallelism, which is nested in world parallelism, is executed by one thread of every process.

However, synchronization using the existing OpenMP constructs is rather expensive for a high amount of nodes

in distributed memory environments. We therefore suggest an additional synchronization mechanism in world parallelism. World memory has two states: uninitialized and initialized. When world memory is allocated, its status is set to uninitialized. The status is changed to initialized as soon as data is written to the memory. In case a process tries to read from unitialized world memory, the reading process or thread gets suspended until another process writes to this memory. The suspended process is reactivated and reads the just written value. This concept requires oversaturating the available nodes with processes to achieve high utilization. The oversaturation may also be used to hide possible memory access latency for data stored at another node. As world memory accesses can be more expensive than local memory accesses, we additionally suggest a parameter similar to `block_local` to automatically prefetch data in node local memory.

Algorithm 2 shows the implementation of the Gauß-Seidel stencil example with world parallelism. `world_parallel` is the keyword similar to `parallel` defining distributed memory parallelism and `omp_world_malloc()` allocates world memory. The iterations of the loop annotated as a world parallel loop are split among all processes. One process per loop iteration is started and each process calculates exactly one loop iteration. Besides the `single` at the beginning of the loop there is no classic OpenMP synchronization, as all synchronization is being done with reading/writing world memory. At first, only the process calculating the first step can be active, as all processes read from `Vs[i]` which is uninitialized for $i \neq 0$. As soon as the first process has calculated e. g. the first tile, the second process can be active and so on. All variables created before the parallel world section are made available as already initialized world memory.

The program shown in Alg. 2 can be easily executed on a multi-core CPU with ignoring any kind of world parallelism, but could also be executed on a cluster of GPUs. Executing on multiple GPUs requires tiling and the parallel execution within the tiles as described in the last section. The synchronization between the GPUs is done using world memory.

Algorithm 2 Gauß-Seidel (distributed memory)

```
1  double** Vs;
2  #pragma omp world_parallel for world(Vs)
3  for (int i=0; i<steps; ++i) {
4    #pragma omp single
5    Vs = omp_world_malloc (iterations);
6
7    Vs[i] = omp_world_malloc (iterations);
8    // in the first iteration copy
9    // the input data from V to Vs[0]
10   if (i==0) memcpy (V, Vs[0]);
11
12   // insert Algorithm 1 and change
13   // the instruction block to
14   Vs[i+1][x+1][y+1] = (Vs[i+1][x][y+1] +
         Vs[i][x+2][y+1] + Vs[i][x+1][y] +
         Vs[i+1][x+1][y+2])/4;
15   // so every iteration writes to its own array
16 }
```

6 Conclusion

Computer systems will change significantly in the upcoming years. Although there will always be a need for developers writing low level code to achieve the maximum performance, development time and productivity are becoming more of an issue. Learning all the details of new hardware and its programming system to rewrite an existing application does not pay off in an increasing set of scenarios, as we are currently seeing an increasing trend in heterogeneity. In this paper we have outlined a proposal of how OpenMP can be extended to support distributed memory and closely coupled processors with manually managed on-chip memory, so the same program could be compiled to a single multi-core CPU, a cluster of GPUs or other distributed and shared memory many-core architectures without the need to rewrite the program. We also expect that existing OpenMP programs could easily be modified with the suggested extensions to be executed on new hardware.

References

[1] OpenMP Application Program Interface, Mai 2008. Version 3.

[2] OpenCL Programming Guide for the CUDA Architecture, August 2009.

[3] LEE, S., MIN, S.-J., AND EIGENMANN, R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2009), ACM, pp. 101–110.

[4] The OpenCL Specification. Version 1, revision 43, May 2009.

[5] OpenCL Development Kit for Linux on Power. `http://www.alphaworks.ibm.com/tech/opencl/`.

[6] PFISTER, G. F. *In search of clusters (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[7] STUART, J. A., AND OWENS, J. D. Message passing on data-parallel architectures. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 1–12.

[8] WOLFE, M. J. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.