View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UPCommons

Scaling Irregular Array-type Reductions in OmpSs

Jan Ciesko[1], Sergi Mateo[1], Xavier Teruel[1,2], Vicenç Beltran[1],
Xavier Martorell[1,2], Rosa M. Badia[1,2] and Jesús Labarta[1,2]
[1]Barcelona Supercomputing Center
[2]Universitat Politècnica de Catalunya
{jan.ciesko, sergi.mateo, xavier.teruel, vicenc.beltran, xavier.martorell, rosa.m.badia, jesus.labarta}@bsc.es

Abstract *– Array-type reductions represent a frequently occurring algorithmic pattern in many scientific applications. A special case occurs if array elements are accessed in a non-linear, often random manner, which makes their concurrent and scalable execution difficult. In this work we present a new approach that consists of language- and runtime support to facilitate programming and delivers high scalability on modern shared-memory systems for such irregular array-type reductions. A reference implementation in OmpSs, a task-parallel programming model, shows promising results with speed-ups up to 15x on the Intel Xeon processor.*

I. Introduction

Irregular array-type reductions, also referred to as scatter-update, represent memory updates over an array type. The non-atomic operation as well their dynamic memory access pattern make their concurrent execution non-trivial and require careful handling to achieve scalability and correctness. Fig. 1 shows a scalar, regular and irregular array type reduction over *target* where in case of an irregular array-type reduction, the update positions depend on indexes generated by a function *f*.

It becomes obvious that algorithms containing an irregular

```
loop construct over i
{
   target = op(target, RHS);
}

loop construct over i
{
   target[i] = op(target[i], RHS);
}

loop construct over i
{
   j = f(i);
   target[j] = op(target[j], RHS);
}
```

Figure 7: Different types of reductions require different parallelization techniques

array-type reduction are cache inefficient due to distant memory accesses and consequently execution performance is bound to the speed of the memory subsystem. Further in order

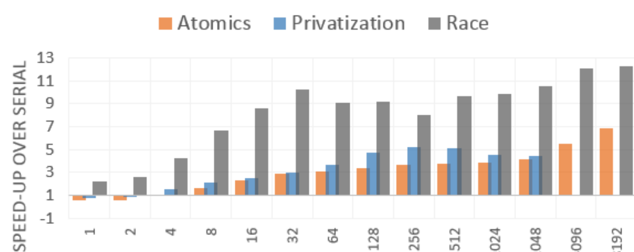


Figure 8: Application scalability of different approaches compared to a plain implementation with data races showing achievable performance on a single MareNostrum node

to avoid a race condition where multiple threads perform an update of a single memory location at the same time, accesses either need to be synchronized (via thread synchronization or memory barriers such as atomics), ordered [1] or redirected [2].

Access redirection to a thread-private copy of the reduction target is a common approach that eliminates the need for access synchronization. While this works well for scalar types, it becomes expensive for arrays and even useless for large data sets.

Figure 2 shows the performance impact of atomics and array privatization in the *RandomAccess* [3] kernel benchmark over serial execution running with 16 threads and different problem

```
for(j = 0; j < num_tasks; j++) {
 INT_TYPE seed = ran[j];
 #pragma omp task concurrent (Table[0;N])
 {
  for( INT_TYPE i = 0; i < N_block; ++i ) {
   seed = LCG_MUL64 * seed + LCG_ADD64;
   INT_TYPE pos = seed >> (bitSize - logTableSize);
   #pragma omp atomic
   Table[pos] ^= seed;  --> __sync_xor_and_fetch_8(&Table[pos], __seed);
}}}
#pragma omp taskwait
```

Figure 9: RandomAccess implemented with atomics, showing the compiler generated instruction that triggers a memory barrier

sizes. Its source code is shown in Figure 3. Consequently a new approach is needed that improves cache efficiency, reduces lock contention, eliminates memory barriers and is applicable on large input data sets at the same time. It turns out that by redirecting accesses to an array of thread-local linear buffers to temporarily store memory updates of a certain memory region of the reduction array and to flush the buffers when they are full is a simple yet efficient technique to meet the above requirement. We present this approach in more detail in the next chapter.

II. Runtime Support

To support irregular array-type reductions in OmpSs, we developed a new approach called Privatization with In-lined Block-ordered Reductions - *PIBOR*. In this approach all memory accesses to the original reduction array are redirected to a thread-private buffer. While this is comparable to regular privatization, the buffer is filled linearly, is limited to a pre-set size and additionally stores the memory address along the data of each access. Once the buffer is filled up, the owning thread reduces the buffer to global memory.

Typically writing out data to global memory requires to perform a global lock over the entire data structure which serializes execution. We prevent this by assigning buffers to discrete memory regions of the reduction array. In this case accesses to the original reduction array in a certain region are stored in the correspond buffer. In case the buffer runs full, the owning thread tries to acquire a lock that protects only the particular memory region of the global array. Buffers corresponding to different regions can now be reduced in parallel and by increasing the number of regions, the effect of lock contention over a single region can be efficiently mitigated. A schematic overview of an application that runs *N* tasks on *N* threads and performs a reduction over an array divided into *M* locations is shown in Figure 4.

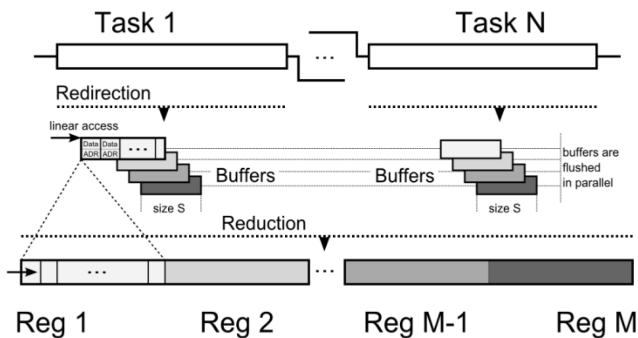


Figure 10: A schematic view of PIBOR showing thread-local buffers and their corresponding array regions

Since buffers correspond to different regions, each memory access needs to be inspected in order to determine its correct buffer. We do so by applying a hash function on the address of the accessed element. The entire process is shown in Figure 5.

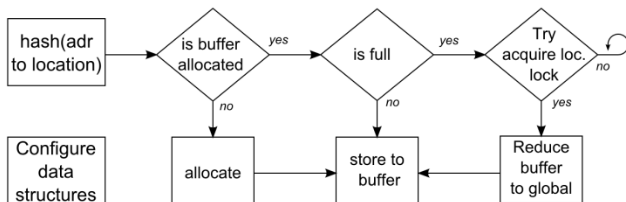


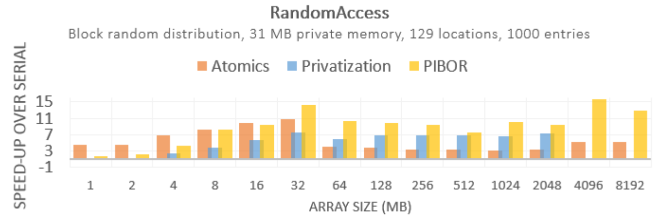
Figure 5: Execution diagram showing the process of privatization and reduction to global data

III. Language support

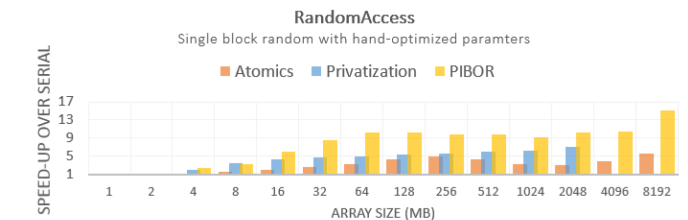
High programmability while maintaining execution transparency is a key requirement for modern programming models. Since PIBOR is conceptually related to privatization, an approach often found in declarative programming models such as *OpenMP* [4], its introduction puts minimal effort on front-end compilers, current specifications and user understanding. The following shows language support for array reductions in OmpSs and its compiler generated code.

```
#pragma omp task reduction (array[0;N])
{                                      ...
   array[pos1] ^= RHS;                 T * __tmp;
   array[pos2] ^= RHS;    ----->       RT.request( & array[posN], __tmp)
   ...                                 *__tmp = RHS;
}                                      ...
```

IV. Case studies



We evaluate the presented approach using RandomAccess on a single MareNostrum 16-way SMP node. RandomAccess is a kernel benchmark that allows to simulate different access patterns. In particular we looked at three representative scenarios: uniform random distribution, block random distribution and single block random distribution where all accesses are restricted to a single memory region. Performance



results are shown in Figure 6 and 7.

Figure 6: PIBOR requiring only 31 MB to achieve speed-ups of up to 15x on a SMP node. Block random distribution favors atomic updates for small (cacheable) problem sizes due to reduced impact of memory barrier and no cache-line invalidations

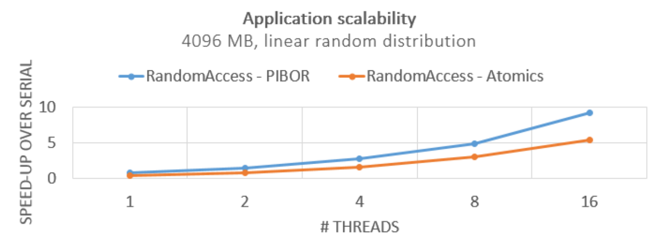


Figure 7: Performance scalability of PIBOR compared to atomics

V. Conclusion and Future work

The presented approach scales by redirecting previously random memory accesses of a region into a linear buffer. Since each buffer corresponds to a memory region of the reduction array, buffers can be flushed in parallel. Further work is directed towards automated tuning of location granularity and buffer sizes and experiments on different processors including *Xeon Phi* and *Power8*.

Acknowledgment

I would like to thank all my coauthors for their invaluable insights and their patience when exposed to my ideas during countless meetings.

References

[1] H. Yu and L. Rauchwerger, Adaptive Reduction Parallelization, 14th ACM Intl. Conf. on Supercomputing, 2000.

[2] H. Han and C.-W.Tseng, A Comparison of Parallelization Techniques for Irregular Reductions, 15th IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS'2001)

[3] RandomAccess, HPC Challenge, http://icl.cs.utk.edu/projectsfiles/
hpcc/RandomAccess, 2015

[4] OpenMP, Application Programming Interface, www.openmp.org, 2015