The OmpSs Reductions Model and how to deal with Scatter-Updates

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 *– Scatter-updates represent a reoccurring algorithmic pattern in many scientific applications. Their scalable execution on modern systems is difficult due to performance limitations introduced by their irregular memory access pattern that prohibits an efficient use of the memory subsystem. Further performance degradation is caused by techniques that are required in order to eliminate potential data races and come at the cost of overhead. Taking a closer look at algorithmic properties, access patterns and common support techniques reveals that a one-size-fits-all solution does not exist and solutions are needed that can adapt to individual properties of the algorithm while maintaining programming transparency. In this work we propose a solution framework that supports a broad set of techniques, provides the required access pattern analytics to allow dynamic decision making and shows what language extensions are needed to maintain programming transparency. A reference implementation in OmpSs, a task-based parallel programming model, shows programmability and scalability of this solution.*

I. Introduction

The widening gap between processor and memory speeds periodically brings up the discussion on how to improve scalability of algorithms that hit the memory wall exceptionally fast due to their scattered memory updates. At the core of the problem are high memory access latencies that become dominant as a result from the caching and bandwidth inefficiencies of these algorithms and the overheads introduced by techniques that ensure correctness by eliminating the possibility of data races. Among these techniques, only a single generally applicable solution exists, namely access synchronization. Synchronization uses software and hardware assisted techniques to implement atomicity of the update operation (read-modify-write) with overheads that differ between processor architectures. Synchronization constructs are typically members of either the language or runtime specification of a programming model and therefore easy to use but unfortunately do not

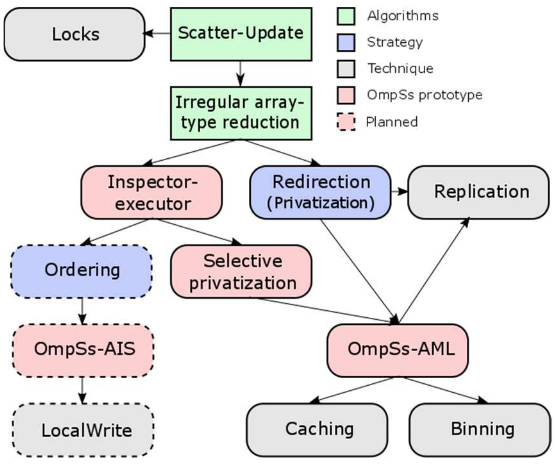

Figure 1 Landscape of algorithms, strategies and techniques

```
while(simulation_runs()){
  #pragma omp loopstep
  #pragma omp task reduction (+:v:SPB) invariant (v)
  for(int i = 0; i < iters; i++) {
     j = f(i);
     v[j]++;
  }
}
```

Figure 2 Reduction kernel with proposed clauses to support alternative memory layouts with inspectors and executors

address the issue of poor locality of these algorithms.

A special case occurs when the iterative scatter-update implements a function that is associative, communicative and has no control dependency between iteration loops (algebraic monoid). These algorithms are called reductions and allow a whole set of additional techniques to improve performance and scalability. Main implications of these properties are two-fold: firstly, the order of memory accesses does not matter anymore which allows concurrent executions without maintaining a constant execution order (of tasks, loop iterations or particular instructions) and the existence of the neutral element allows the use of scratch data to temporarily store intermediate results. This led to the development of different support techniques [1] that fall into two strategies. Access redirection is a strategy where accesses are redirected to a scratch storage while leaving the iteration space untouched. The scratch memory is typically a thread-private copy of the original data (replication) or any data structure that fulfills a similar goal. Ordering is another strategy that avoids redirection and reorders iterations by specific criteria instead. Which of these is used and how they are configured depends on algorithmic properties that require both compiler support and runtime analysis. As of today, none of these techniques other than replication made its way into popular parallel programming models.

In this work we present the OmpSs Reductions Model (OmpSs-RM) which implements a framework to support redirection techniques with alternative memory layouts (OmpSs-AML) such as binning or software caching as well as ordering techniques that require alternative iteration spaces (OmpSs-AIS) such as LocalWrite in near future. In particular we show what new language constructs are needed, how the inspector-executor model can be integrated into the runtime as well as how scientific applications can benefit from these techniques. Figure 1 shows a landscape of algorithms, strategies, techniques and their support in OmpSs.

II. Support in OmpSs

The OmpSs-AML implementation builds on top of the existing functionality of reduction scope definition, pre-allocation, allocation on demand and lazy initialization. In order to support AMLs, we require three additional information from the developer.

Firstly, the developer is required to express the intention to use an AML. This step is necessary in order to preserve consistency as with AMLs, the scratch memory is not necessarily a replica of the original data anymore. For this purpose, we prose the extension of the *reduction clause* by the additional parameter *MODE*, where mode is an identifier of a vendor provided privatization technique.

Further, in order to support AMLs that require an inspector-executor, we propose the addition of the *invariant (target)* clause. The invariant clause defined over a *target* specifies that the access pattern of the target as well as the calling order within the scope of a reduction are invariant. This step is important to guarantee that the inspector-executor is always applied to the matching function and that optimization results obtained during the inspection phase are still valid for subsequent function calls or task instances.

```
while (...){
  ID = instance_invariant_identifier;
  frameID = handle_optimization_frame(ID)
  task = new reduction_task(frameID, taskcode ,...)
  task.run();
}
...
taskcode (...){
  analytics * a; AML * aml;
  t frameInstanceID = get_frameInstanceID();
  aml = get_thread_storage(v);
  analytics = get_analytics(frameInstanceID);
  if(!analytics.ready)
    inspect(&v[j],analytics ,frameInstanceID);
  for (...) {
    j = f(j);
    (*SPB_get(&v[j], analytics , aml))++;
  }
}
```

Figure 3 Intermediate code prototype showing the main loop body, task code and runtime APIs

Lastly we propose the addition of the *loopstep* pragma. This pragma defines the scope of an optimization frame and is used to differentiate between inspection and execution phases. Figure 2 shows high-level code that uses selective privatization and an AML to implement an array-type reduction.

Figure 3 shows an intermediate code prototype where an instance-invariant identifier (*frameID*) is created to identify an optimization frame and that is subsequently passed to all participating task. By doing so, all tasks sharing one identifier are associated to one optimization frame. Once a task instance is created, the frame identifier is used to generate a new unique identifier for that particular task instance (*frameInstanceID*). In OmpSs and for the context of reductions, the frame identifier is computed as XOR between the reduction target address and value of the reducer function pointer.

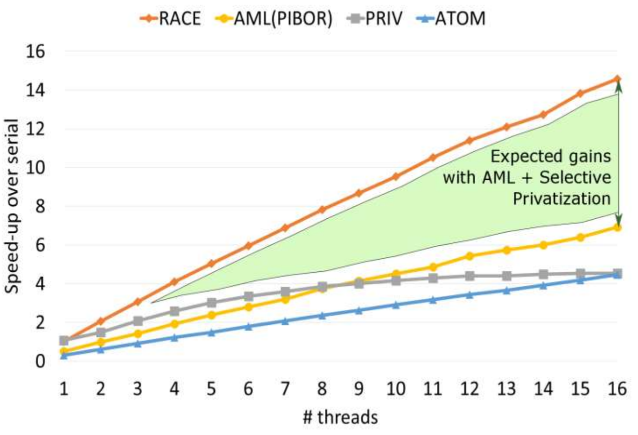

Figure 4 Lulesh reduction kernel scalability on the Xeon E5 processor with different support techniques showing the expected performance when properly exploiting access locality of tasks with AMLs and selective privatization

The instance frame identifier for each particular task is created again as an XOR between frame identifier and a task creation counter. In case of nesting, new identifiers are created for each nest. Currently, optimization frames across nesting levels are not supported.

III. Case study

Our work on OmpSs AML and the inspector-executor model was largely motivated by Lulesh, a seismic simulation code that contains irregular array-type reductions. Inspecting its memory access pattern revealed a linear access pattern with very small overlaps for boundary iterations. The inspector used in this case records histograms over addresses, over distances between memory accesses and over the rate of distance changes. This information is evaluated once the optimization frame is completed. For the case of Lulesh and AML with selective privatization, the evaluation produces an ownership table that is used in the executor phase to determine whether an update operation accesses task local data or not. Since most accesses in Lulesh are local, the original data can be updated without the need of synchronization nor redirection. Figure 4 shows scalability of a Lulesh reduction kernel implemented with different techniques. We expect that inspector-executor enabled AMLs will be close to a version that is free of any additional overheads but contains data races (RACE). Further evaluation is pending.

IV. Conclusion and Future work

We are currently evaluating OmpSs AMLs and inspectors-executors to derive further knowledge about overheads of the inspection phase, its usability in other applications and architectures as well as the integration of alternative iteration spaces (AIS) into OmpSs. This work aims to influence the OpenMP specification to support this type of algorithms in the future.

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