UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

On Algorithmic Reductions in Task-parallel Programming Models

Jan Ciesko

Doctoral Thesis
Computer Architecture Department
Universitat Politècnica de Catalunya

Director: Rosa M. Badia
Co-director: Jesús Labarta Mancho

Barcelona, Spain 2017

In collaboration with

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Abstract

Wide adoption of parallel processing hardware in mainstream computing as well as the interest for efficient parallel programming in developer communities increase the demand for programming models that offer support for common algorithmic patterns.

An algorithmic pattern of particular interest are reductions. Reductions are iterative memory updates of a program variable and appear in many applications. While their definition is simple, their variety of implementations including the use of different loop constructs and calling patterns makes their support in parallel programming models difficult and requires a careful design for programmability, transparency and performance. Further, their characteristic update operation over arbitrary data types that requires atomicity makes their execution computationally expensive and scalable execution challenging. These challenges and their relevance makes reductions a benchmark for compilers, runtime systems and hardware architectures today.

Driven by our own demand for their efficient support in the OmpSs parallel programming model, we have developed new ideas and features that we present in this work. Our contributions are as follows: first, we add support for task-parallel reductions (for *while*-loops and generally recursive functions) in the OmpSs programming model and develop a proposal for their inclusion in the OpenMP specification. Second, we develop new software techniques to accelerate irregular and near-regular array-type reductions and evaluate their impact with different applications on different hardware architectures. Third, we show how these techniques can be supported in OmpSs and OpenMP; and fourth, we show that reductions benefit from smart runtimes implementing an inspector-executor and that this execution model can be integrated into a task-parallel programming model. Our proposal for task-parallel reductions has been recently accepted into the OpenMP standard.

Resumen

La amplia adopción de hardware de procesamiento paralelo para la computación de propósito general, así como el interés por una programación paralela eficiente en la comunidad de desarrolladores, han aumentado la demanda de modelos de programación que ofrezcan soporte para patrones algorítmicos comunes.

Un patrón algorítmico de particular interés son las reducciones. Las reducciones son actualizaciones iterativas de memoria de una variable del programa y aparecen en muchas aplicaciones. Aunque su definición es simple, su variedad de implementaciones, incluyendo el uso de diferentes construcciones de bucle y patrones de llamada, hace que su soporte en los modelos de programación paralelos sea difícil y requiera un cuidadoso diseño en lo que respecta a programabilidad, transparencia y rendimiento. Además, la necesidad de atomicidad en la ejecución de estas operaciones hace que sean costosas desde el punto de vista computacional y difícilmente escalables. Estos desafíos y su relevancia convierten a esta clase de operaciones en una referencia para medir el rendimiento de compiladores, sistemas en tiempo de ejecución y arquitecturas de hardware actuales.

Impulsados por la necesidad de disponer de una implementación eficiente en nuestro modelo de programación paralelo, hemos desarrollado nuevas ideas que presentamos en este trabajo. Nuestras contribuciones son las siguientes: en primer lugar, añadimos soporte para reducciones de tareas paralelas (para bucles *while* y funciones recursivas) en el modelo de programación OmpSs y desarrollamos una propuesta para su inclusión en la especificación de OpenMP.

En segundo lugar, desarrollamos nuevas técnicas para acelerar las reducciones irregulares y casi-regulares de tipo *array* y evaluamos su impacto mediante diferentes aplicaciones en varias arquitecturas.

En tercer lugar, mostramos cómo estas técnicas pueden ser soportadas en OmpSs y OpenMP. Asimismo, mostramos que las reducciones se benefician de sistemas en tiempo de ejecución inteligentes implementando un esquema inspector-ejecutor.

Nuestra propuesta de reducción de tareas paralelas ha sido aceptada recientemente en el estándar OpenMP.

Acknowledgements

We thankfully acknowledge the support of the European Commission through the ENCORE project (FP7-248647), the TERAFLUX project (FP7-249013), the TEXT project (FP7- 261580), and the HiPEAC-3 Network of Excellence (FP7/ICT 217068), further the support of Intel-BSC Exascale Center, the Spanish Ministry of Education (TIN2007-60625, TIN2012-34557, TIN2015-65316-P and CSD2007- 00050) and the Generalitat de Catalunya (2009-SGR-980).

Contents

List of Figures

List of Publications

[P1] Jan Ciesko, Sergi Mateo, Xavier Teruel, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. **Supporting Adaptive Privatization Techniques for Irregular Array Reductions in Task-parallel Programming Models.** *12th International Workshop on OpenMP (IWOMP 2016)*, Nara, Japan: Springer, p. 336-349, 2016.

[P2] Jan Ciesko, Sergi Mateo, Xavier Teruel, Vicenç Beltran, Xavier Martorell, and Jesús Labarta. **Boosting Irregular Array Reductions through In-lined Block-ordering on Fast Processors.** *19th IEEE High Performance Extreme Computing Conference (HPEC15)* , Waltham, USA, IEEE Explore, 2015. (Best Paper Candidate)

[P3] Jan Ciesko, Sergi Mateo, Xavier Teruel, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, Alex Duran, Bronis R. de Supinski, Stephen Olivier, Kelvin Li, and Alexandre E. Eichenberger. **Towards Task-parallel Reductions in OpenMP.** *11th International Workshop on OpenMP (IWOMP 2015)*, Aachen, Germany: Springer, p. 189-201, 2015.

[P4] Jan Ciesko, Sergi Mateo, Xavier Teruel, Vicenç Beltran, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. **Task-parallel Reductions in OpenMP and OmpSs.** *10th International Workshop on OpenMP (IWOMP 2014)*, Salvador, Brazil: Springer, pp 1-15, 2014.

[P5] Jan Ciesko, Javier Bueno, Nikola Puzovic, Alex Ramirez, Rosa M. Badia, and Jesús Labarta. **Programmable and Scalable Reductions on Clusters.** *27th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2013)*, Boston, United States, IEEE Explore, 2013.

Chapter 1

Introduction

Reductions are iterative updates of a program variable, a simple algorithmic pattern with significant impact keeping computer scientists busy for over two decades. Looking back it turns out that the majority of research conducted on this matter coincides with the advent of parallel architectures in mainstream computing and the rising demand for programming productivity on these systems. It corresponds to the time where energy constraints shifted advancements to increase processor performance from frequency scaling towards parallel hardware and where programmers were becoming ready to dedicate some of the processor's cycles to run more advanced programming models. Their use promises higher development efficiency and performance portability.

As a result, concurrent algorithms and parallel programming models gained attention and also moved reductions into the spotlight. Unfortunately, soon it became clear that reductions do not achieve a satisfactory efficiency on parallel hardware. While the update loops can be parallelized easily, data is shared among processors. This results in caching effects that impact scalability. Further, data races may occur during parallel execution compromising correctness.

Consequently, reduction support in software and hardware is desired with the common objective of delivering scalable performance while maintaining correctness of code. In case of software implementations, support in parallel programming models alleviates the programmer of implementing optimization techniques manually. Hardware support accelerates software implementations

```
var r = 0;
for(var i=0; i<8; i++)
{
  r++;
}
```

Figure 1.1: A commonly occurring reduction kernel performing iterative updates in a for-loop over a scalar data type

at the cost of additional hardware complexity. The extent of complexity is to be determined in future work.

In software, designing language support and integrating advanced runtime techniques into parallel programming models are the two key challenges that are being faced in academia and industry today. Approaching these challenges in the context of task-parallel programming models is the objective of our work. Figure 1.1 shows a simple case where the reduction variable r of a scalar type is iteratively updated in a *for-loop* - the only reduction pattern supported in OpenMP [1] at the time we started work on this topic.

While researching programming model support for reductions we identified five particular areas of interest:

- Challenges for scalability and correctness
- Current and future software techniques
- Challenges to support irregular access patterns
- Challenges to support near-regular access patterns
- Language support and programmability

The following sections provide more context for each area.

1.1 Challenges for scalability and correctness

Maintaining correctness is a fundamental property, therefore approaches are needed that are race condition free while offering scalable performance on

```
...
for( Index_t k=0 ; k<numElements ; ++k )
{
  const Index_t* const elemToNode = domain.nodelist(k);
  Real_t fx_local[8] ;
  Real_t fy_local[8] ;
  Real_t fz_local[8] ;

  CollectDomainNodesToElemNodes(fx_local,fy_local,\
                                              fz_local,...);
  Computation1(fx_local,fy_local,fz_local,...);
  ...
  ComputationN(fx_local,fy_local,fz_local,...);

  for( Index_t lnode=0 ; lnode<8 ; ++lnode ) {
    Index_t gnode     = elemToNode[lnode];
    domain.fx[gnode] += fx_local[lnode];
    domain.fy[gnode] += fy_local[lnode];
    domain.fz[gnode] += fz_local[lnode];
  }
}
...
```

Figure 1.2: IntegrateStressForElems(), a reduction kernel from the LULESH hydrodynamics application defined over array-type variables with an irregular access pattern, representing a challenging use-case

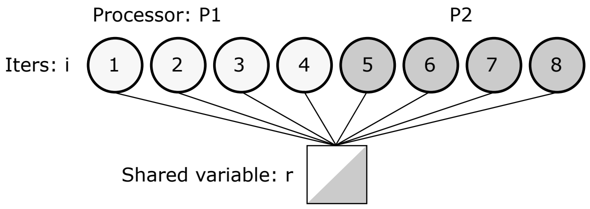
current parallel architectures. To eliminate race conditions, two fall-back strategies exist: atomic updates and privatization.

Atomic updates guarantee atomicity of the read-modify-write cycle and are generally well suited for low-contention situations where the atomic update is invoked infrequently. Unfortunately for the opposite case, execution performance depends on the underlying architectural support. This results in low performance portability between systems. Further, sharing of data between processors results in coherence traffic and higher access latencies. Together with the fact that only a subset of common data types is supported, requiring mutexes for the generic case, their use turns out to be less favorable in parallel programming models.

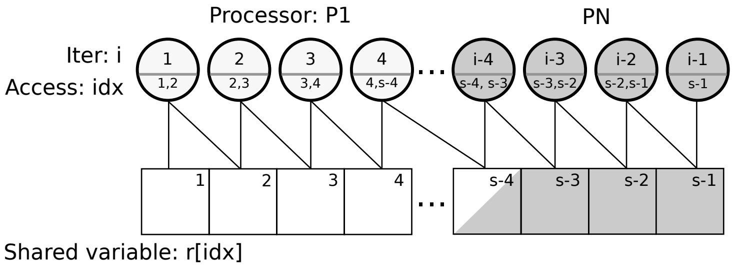
To avoid overheads associated with atomic accesses, the popular approach of privatization can be applied. Privatization, also called replication, describes the idea of redirecting memory access into thread-private copies of the original variable. The redirection is typically generated by a parallelizing compiler where occurrences of the reduction variable are syntactically replaced by references to thread-private copies. The privatization step, that is allocation of the thread-private data and its initialization, is followed by a hierarchical and parallel merge phase once the computation is complete. Privatization is the *de facto* standard with implementations reaching across programming models and hardware architectures today. It gained popularity as this approach supports reductions over small data (as being the case in Figure 1.1) reasonably well, keeping overheads of allocation, initialization and reduction of thread-private copies small. Further, this approach does not require complex code transformations, making support in parallelizing compilers relatively straightforward. Unfortunately, for larger data (array types) and for higher numbers of processors, privatization does not scale. Figure 1.2 shows a reduction kernel from the LULESH hydrodynamics simulation code [2]. It represents a particularly challenging use-case where a reduction is defined over three arrays, *fx*, *fy* and *fz* of type *float*, each accessed in an irregular pattern as defined by *elemToNode*. To support array-type reductions, neither atomic access nor replication are suitable approaches.

Consequently a wide body of research has been conducted on more advanced parallelization strategies with projected implementations in both hardware and software. Among these techniques only a small fraction gained relevance. The lack of hardware support can be mainly attributed to the fact that established processor functionality such as atomics or access synchronization cover certain scenarios sufficiently well which together with the complexity of adding new features discourages hardware manufacturers to invest into further development. A thinkable hardware support may include extensions to the processor ISA to relax cache coherence for cache lines that correspond to reduction variables as well as offloading computation to functional units located on memory controllers, thus implementing a push semantic and avoiding data round trips. Still, the question arises if software-based techniques are not sufficient enough making reduction support a matter of software support after all.

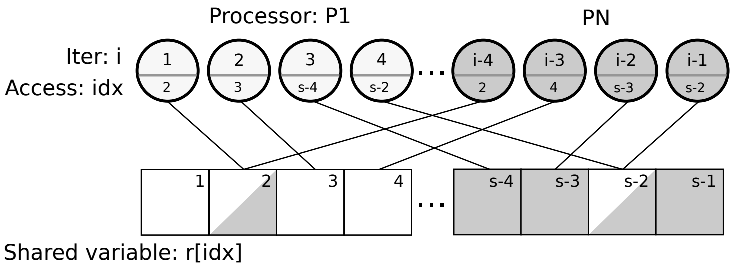
Figures 1.3(a) and 1.3(b) show representations of memory access patterns for codes in Figures 1.1 and 1.2. It illustrates the distribution of iterations between processors represented by circles, memory accesses as edges and



(a) Memory access pattern of a concurrent scalar-type reduction over the variable r



(b) Memory access pattern of a concurrent near-linear reduction defined over an array-type variable r with one element being updated by two processors. It represents a simplified version of the memory access pattern of the reduction kernel shown in 1.2



(c) Memory access pattern of a concurrent, irregular array-type reduction defined over an array-type variable r

Figure 1.3: Representative memory access patterns of scalar and array-type reductions showing conflicting accesses resulting from parallel execution by multiple processors

accessed memory locations as rectangles. Figure 1.3(c) shows the memory access pattern for irregular array-type reductions. We discuss application with this access pattern later in this work.

1.2 Current and future software techniques

While synchronized access or privatization works well for scalar types, solutions are needed to cover reductions over arrays. Several advanced techniques have been proposed among which *SelectivePrivate* [3] and *LocalWrite* [3] gained most significance and also influenced our work. Selective privatization uses a conditional redirection of accesses into thread-private data replicas. LocalWrite is a technique where memory updates remain unchanged but the iteration space for each thread is modified in order to avoid data races. Both techniques are suited for cases where the application exhibits a particular memory access pattern, a property which limits their general applicability. However, these techniques gave inspiration for our work towards generalization and deriving the following classification.

In our work, we group techniques by their underlying approach into three categories:

- **Reductions with direct access.** This strategy relies on atomicity or equivalent implementation of the read-modify-write operation. An example of this technique is the use of atomics.

- **Reduction support through access redirection.** This group covers techniques that use the idea of access redirection into thread-private objects. This eliminates potential race conditions and caching effects due to sharing. Replication, as discussed in the previous section, is therefore a special case of access redirection where the thread-private object is a copy of the reduction variable. A generalization of this approach can be achieved if the original memory updates are replaced by a function *f(address, value, parameters)*. It is left to the programming model vendor to implement *f* following any particular placement strategy including conditional redirection. Consequently the allocated thread-private storage is implementation dependent and we refer to it as thread-private data container. Techniques within this category are suited for algorithms with irregular memory accesses where the placement function can improve locality and where the cost of privatization can be amortized. SelectivePrivatization is an approach that falls within the category.

- **Reduction support through ordering.** This covers techniques that avoid race conditions and shared accesses by assigning particular iterations of the reduction loop to each thread. LocalWrite is an example for this category. Techniques following this strategy are typically suited for reductions with sparse or near-regular memory access pattern as in this cases, cost of ordering is low.

We call techniques that redirect accesses into thread-private data containers by implementing the placement function f as reductions with *alternative memory layouts*, *AML*, and techniques that order iterations as reductions with *alternative iteration spaces*, *AIS*.

It is interesting to point out that integration of advanced approaches into existing programming models has not been carried out successfully yet. As a result these techniques are rarely used leaving cases like that shown in Figure 1.2 unsupported. We believe that by abstracting from particular techniques, a generic support in parallel programming models can be achieved. We show how techniques with redirection and ordering can be supported transparently in task-parallel programing models in this work.

1.3 Challenges to support irregular access patterns

Irregularity is a dynamic property of an algorithm that results in a random sequence of accessed memory locations, an access pattern that is inherently cache inefficient. Irregular array-type reductions exhibit such an access pattern where seemingly random array positions are updated. Supporting these cases is non-trivial as conventional techniques of atomic access and privatization do not improve cache performance. With these techniques, the sequence of accessed memory addresses remains unchanged. It turns out that irregular array reductions can be well supported by techniques with access redirection that implement a data placement strategy for improved cache efficiency.

Taking a closer look at techniques with access redirection reveals that four design features are important for scalable performance:

- Avoidance of data replication (creating full data copies) for non-scalar data types
- Avoidance of privatization for unrelated threads
- Support for custom data placement strategies (by allowing different implementations of f)
- Support of arbitrary thread-private data containers (objects)

In this work we propose two new approaches that fulfill these requirements. One implements access redirection into software caches to support architectures with high memory latencies (clusters), also called *CachedPrivate*; and binning, an approach that uses a fast hash function to sort accesses into bins corresponding to memory regions for higher data locality. We also refer to binning as *PIBOR*, *Privatization with In-lined Block Ordering.* Both, software caching and binning are examples of reduction techniques with redirection, implementing alternative memory layouts and data placement strategies. These techniques achieve the highest speed-ups compared to replication and atomic access. A schematic representation of an irregular access pattern is shown in Figure 1.3(b).

We discuss these techniques in chapters 5 and 6. The generalized support is discussed in Chapter 7.

1.4 Challenges to support near-regular access patterns

A special case occurs when a seemingly irregular memory access pattern turns out to be sparse or near-regular. The latter typically occurs when the reduction kernel is preceded by a sorting phase (e.g. coloring), a technique to reduce irregularity of accesses. In these cases each participating thread accesses either disjoint memory regions or memory regions with small overlaps with other regions accessed by other threads. To support these cases, techniques are needed that benefit from these access properties. In turns out that techniques building on top of an inspector-executor approach are well suited for this purpose. Techniques that build on top of this execution model fall within both aforementioned categories.

From the programming model perspective, the following features are required to support inspector-executor techniques:

- The reduction kernel must be executed multiple times during the lifetime of the application such that the cost of inspection can be amortized.
- The programmer must define a point in the source code where the runtime completes inspection and switches to execution.
- The inspection phase must not change application behavior.
- The runtime must implement task instance identifiers and their tracking across instantiation. This guarantees that inspection results are applied to the same task instance during the next iteration.
- The programmer must be aware of the requirement that the instantiation order of tasks must remain unchanged.

Once access properties are recorded and ownerships defined, the execution phase can switch from inspection to execution. We propose an executor that avoids privatization all together based on dependency-aware task scheduling, called *DepRed*. It is an approach that uses work scheduling based on data dependencies. During inspection, accessed memory regions and their overlaps are identified. During execution, overlapping memory regions are interpreted as dependencies between units of work (tasks). To avoid data races, only tasks with no overlapping regions are scheduled for execution at a time. This approach is well suited for task-parallel programming models that support the expression of data-flows in the code. Since dependency-aware task execution results in a different order of how loop iterations are processed, this approach falls into the category of techniques with alternative iteration spaces (AIS).

This technique yields near-linear speed ups for the reduction kernel of the LULESH application shown in Figure 1.2. In this case the inspector is able to detect a near-regular access pattern (as shown in Figure 1.3(b)) with small overlapping regions. How AIS techniques with an inspector-executor execution model can be supported in task-parallel programming models is discussed in Chapter 7.

It is important to point out that our programming model support for reductions foresees that the programmer is in charge to take a decision on which

```
int foo(...){
  if (check1(...)) return 1;
  int count = 0;
  for (int i=0;i<width;i++){
    if (check2(...)))
      count += foo(...);
  }
  return count;
}
```

(a) Recursive reduction

```
...
int red = 0;
int foo(node_t node,...){
 while(node->next){
   red+=bar(node->value);
   node=node->next;
 }
}
...
```

(b) While-loop reduction

Figure 1.4: Common occurrences that require the support of task-parallel reductions to express concurrency

technique to use. Therefore, in Chapter 7, we discuss a key indicator that hints on the suitability of particular approaches for particular cases.

1.5 Language support and programmability

Language support for common algorithmic patterns is a key property for achieving programming efficiency. In the context of declarative task-parallel programing models, language constructs are needed that capture programmer intention and algorithmic parameters. With this information parallelizing compilers can generate the appropriate code and invoke runtime functionality. At the same time language design is needed that maintains consistency and programmer understanding inherited from the surrounding parallel programming model. For general reduction support, the expression of the following programmer intents must be supported syntactically:

- Declaration of a reduction computation in the context of a parallel region. This applies to programming models that implement the concept of a parallel region. In OpenMP, worksharing constructs support this feature through the *reduction* clause already.

- Declaration of a reduction computation in the context of a *task*. This would allow the support of task-parallel reductions including while-loop programs and recursions. (Unsupported intent)

- Expression to invoke a particular underlying reduction technique. (Unsupported intent)
- Expression of inspector-executor phases. (Unsupported intent)

The three unsupported intents are further explained in the next paragraphs.

Expressing task-parallel reductions Task-parallel algorithms represent a super-class of for-loop computable algorithms. These algorithms include while-loop computable, also called general recursive, algorithms. Further task-parallelism is a convenient way to express concurrency in recursive algorithms. Adding reduction support for this type of algorithms represents an important step towards programmability.

While for-loop reductions were supported in OpenMP from early on, task-parallel reductions remained unsupported. This is due to the following reasons:

- Task-parallel reductions do not define a scope *per se*. For a parallel programming model it is impossible to know when the scope of a reduction ends, a piece of information that is crucial since the runtime is required to invoke functionality corresponding to a given reduction support technique to finalize a reduction.
- The occurrence in recursions rises the demand for reductions support that avoids the step of invoking a runtime technique for each recursive call.
- The specification of OpenMP does not foresee the allocation of a data context in its task specification. In case of reductions, a data context is always required. This collision with existing specification requires non-trivial changes of the OpenMP specification.

Consequently, a solution is needed that allows the definition of scope, supports recursions and minimizes impact on the specification. Our solution presented in chapters 3 and 4 shows a solution that complies with these requirements and was recently accepted into the OpenMP specification.

Figure 1.4 shows examples of a while-loop and recursive reduction.

Expression of technique To enable meaningful support for array-type reductions, syntactical means are needed to allow the developer to express which underlying technique to apply. Once a technique is selected by the programmer, the compiler can perform the appropriate code transformations and invoke the selected runtime support.

Expression of inspector-executor phases The inclusion of inspector-executor based techniques requires the definition of a switch-point. This point in code indicates to the programming model runtime that the inspection phase is completed and execution mode can transfer to an executor.

In Chapter 7 we present language constructs to define a technique as well as to express inspector-executor phases for OpenMP and OmpSs [4].

1.5.1 Why building on top of task-parallelism?

We base our work on declarative task-parallel programming models. For programmers, tasking represents a well understood and easy to use concept of work decomposition. For programming model vendors, it offers freedom to define surrounding concepts and to provide runtime libraries implementing advanced functionality in areas such as scheduling, accelerator- or distributed memory support. A runtime that implements such features is included in the *OMP Superscalar* (*OmpSs*) programming model.

OmpSs is a declarative, task-parallel programming model supporting data-flow based task scheduling. Data-flows are expressed by the programmer using the input and output clauses in the task pragma. While OmpSs supports comparable programming primitives as OpenMP, its execution model is centered around tasks, leaving the concept of threads or parallel regions out of concern. Parallel execution is obtained when tasks become ready as their data dependencies are progressively satisfied and are scheduled for execution. Its proximity to OpenMP makes OmpSs a forerunner for many novel techniques and a suitable experimentation platform for our work.

1.6 Contributions

Our work advances reduction support in OmpSs and OpenMP, two relevant parallel programming models in academia and industry. In particular, we contribute to the adoption of task-parallel and array-type reductions in both programming models. Support for task-parallelism increases programmability by allowing the expression of while-loop and recursive reductions.

For array support, we provide the ground work consisting of language and runtime features for a more generic support. This allows software vendors to implement optimized techniques that follow either the strategy of redirection or of implementing an alternative iteration space. Techniques with redirection are generalized by providing a programmable placement function that stores data into thread-private containers with arbitrary memory layouts. Software caching and binning are two exemplary implementations. Alternative iteration spaces can be implemented through dependency-aware task scheduling but generalization is yet to be determined. Finally we show that runtimes benefit from inspector-executors, an execution model allowing optimizations based on dynamic properties of an algorithm. For evaluation we have selected three representative applications: LULESH, SPECFEM3D [5] and SmartJumper [6]. LULESH and SPECFEM3D are production codes and represent array-type reductions with near-linear access patterns. SmartJumper is based on the popular RandomAccess [7] benchmark and is representative for irregular array-type reductions.

Our contributions can be structured as follows:

1. Support for task-parallel reductions in the OmpSs programming model.
2. Proposal to support task-parallel reductions in OpenMP.
3. Development of *CachedPrivate* - a technique implementing redirection into software caches to accelerate near-regular array-type reductions on distributed memory systems (clusters).
4. Development of *PIBOR* - a software technique implementing address-based binning to support irregular array-type reduction on shared memory systems.
5. We show how support for advanced techniques with access redirection and ordering can be generalized through the ideas of AMLs and AIS. This includes the addition of the inspector-executor and related pragmas in the OmpSs programming model.

6. Evaluation of techniques with LULESH, SPECFEM3D and SmartJumper.

1.7 Thesis organization

In the next chapter we describe the current state of reductions support with popular techniques on current hardware architectures and discuss related work.

The rest of this document is structured as follows. In Chapter 3 we discuss language and runtime support for task-parallel reductions in OmpSs. In Chapter 4 we present our proposal for OpenMP. In Chapters 5 and 6 we introduce techniques to support array-type reductions. In Chapter 7 we discuss support for AMLs and AIS and the inclusion of an inspector-executor in OmpSs. Further, in Chapter 8 we present case studies. Finally, Chapter 9 concludes this topic and previews future work.

Chapter 2

Background and Related work

The promise of parallel processing is performance scalability - a claim that heavily relies on an application's algorithm. One particular group of algorithms that periodically brings up the discussion on how to achieve scalability are concurrent updates of scalar and array-type data. A concurrent update operation contains at least two sequences of read-modify-writes. Due to a non-atomic implementation of this sequence on most hardware architectures, parallel execution results in data races. Data races manifest themselves as altered program behavior relative to serial execution (and typically relative to the programmer's expectations). Updates over array-types have similar correctness properties but are defined over array-types. Both, scalar as well as array-type updates require techniques that ensure correctness.

Among techniques to support concurrent updates only a single generally applicable solution exists, namely access synchronization. Synchronization uses software and hardware assisted techniques to implement atomicity. Synchronization constructs are often members of either the language or runtime specification of a programming model and therefore easy to use but come at a cost of execution overhead.

A special case occurs when the update operation implements an iterative function that is associative, commutative and has no control dependency between loop iterations. These algorithms are called reductions and allow a whole set of additional techniques to improve performance and scalability.

Thread 1	Thread 2	Access	Data Value
			0
READ		↑	0
MODIFY			0
WRITE		↓	1
	READ	↑	1
	MODIFY		1
	WRITE	↓	2

(a) Sequential

Thread 1	Thread 2	Access	Data Value
			0
READ		↑	0
MODIFY	READ	↑	0
WRITE	MODIFY	↓	1
	WRITE	↓	1
			1
			1

(b) Parallel

Table 2.1: Without proper synchronization, parallel updates implemented as read(↑), modify and write (↓) sequences, produce data races on current processors. This results in altered program behavior (results)

In programming, a reduction occurs when a variable, *var*, is updated iteratively as

$$iter: \;\; var = op(var, expression), \tag{2.1}$$

where *op* is a commutative and associative operator performing an update on *var* and where *var* does not occur in *expression*. An array-type reduction over the variable $\overrightarrow{var}$ is defined as

$$iter\ i: var[i] = op(var[i], expression), \tag{2.2}$$

where *i* is an induction variable, *iters* is an iteration space, and where *v* is a reduction variable with *op* being an algebraic monoid.

Similarly, an irregular array-type reduction is defined as

$$\begin{aligned} &iter\ i: \\ &\qquad j = f(i); \\ &\qquad var[j] = op(var[j], expression) \end{aligned} \tag{2.3}$$

with *f* implementing a function that returns integers in the range of zero to number of addressable elements in $\overrightarrow{var}$.

Such irregular array-type reductions appear in FEM solvers where simulation data is updated over irregular mashes, in n-body simulations as well as other graph algorithms where edges connect vertices and processing of each vertex results in at least two irregular memory updates.

Implications of the mathematical 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 or loop iterations and the existence of the neutral element allows the use of scratch data to temporarily store intermediate results. These properties led to the development of different support techniques among which atomic updates and data replication are the most commonly used. Their objective is to eliminate potential race conditions while trying to keep associated overheads small. Table 2.1 shows sequences of read-modify-writes with two potential execution orders. An undetermined program behavior is called data race or race condition.

2.1 On Atomics and Data Replication

The use of atomic updates is perhaps the most straight-forward solution to avoid data races. As the name suggests, atomic updates make the sequence of read-modify-write appear as one atomic operation. Implementations differ. Lock based atomicity relies on an atomic compare and swap instruction (*__sync_val_compare_and_swap*). This instruction represents the basic building block to implement any locking strategy. Other instructions exist that offer more functionality such as the atomic addition (*__sync_fetch_and_add*) or subtraction (*__sync_fetch_and_sub*). However, the underlying implementation in hardware differs substantially with variations in performance.

IBM's POWER8 processor implements atomic via a pair of load-link and store-conditional instructions. This solution is lock-free as a load operation returns the value in memory but a store only succeeds if no update has occurred by another processor since the last load. If the value was updated, a store miss triggers a reload from memory.

On Intel Xeon E5, atomics are implemented with locks on cache-line granularity. In addition, locks enforce in-order execution which affects data prefetching. Taking a look at the GCC manual reveals the following: "In most cases, these builtins are considered a full barrier. That is, no memory operand will be moved across the operation, either forward or backward. Further, instructions will be issued as necessary to prevent the processor from speculating loads across the operation and from queuing stores after the operation." [8]. On

```
while(condition())
{
  #pragma omp atomic
   a++;
}
```

(a) Sample code

```
...
  lock; addl $1,-4(%rsp)
...
```

```
...
.retry:
  lwarx 9,0,10
  addi 9,9,1
  stwcx. 9,0,10
  bne- .retry
...
```

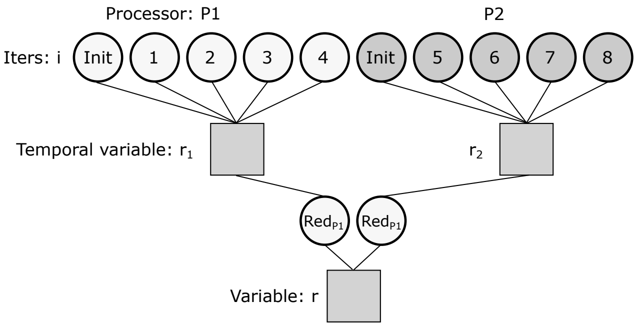
(b) Assembly codes on Intel's x86 and IBM's POWER8 ISAs (right)

Figure 2.1: Different ISAs implement atomics differently. While the x86 ISA issues a global barrier, the POWER8 ISA (on the right) implements a conditional write.

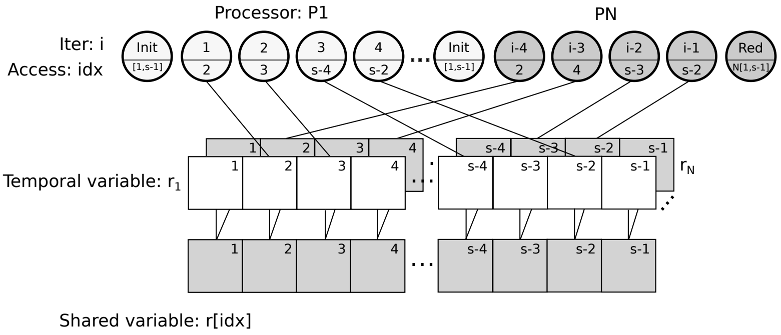
Intel Xeon Phi, atomics have a smaller impact since the processor implements in-order execution only.

Figure 2.1 shows the implementations on assembly level of an atomic increment on the Intel x86 and IBM POWER8 ISAs produced by the GCC compiler on each platform. While the x86 implements a memory barrier (*lock*), the POWER8 uses a conditional store *stwxc*. If the store fails, the operation is repeated. Further comparison of synchronization primitives was conducted by David, Guerraoui and Trigonakis [9].

The other common technique is replication. Replication avoids data races by redirecting memory accesses to thread-private data copies and is typically implemented by parallelizing compilers for scalar-type reductions. Data privatization is efficient in cases where the cost of allocation, initialization and reduction of data is sufficiently small relative to the computation, and performance and applicability are not limited by memory bandwidth (that is if data fits into cache) or physical memory size. Unfortunately, with growing core counts per processor as well as higher numbers of threads per core (SMT), privatization of even small arrays becomes expensive as the algorithm becomes memory bound or even unusable due to memory size limitations.

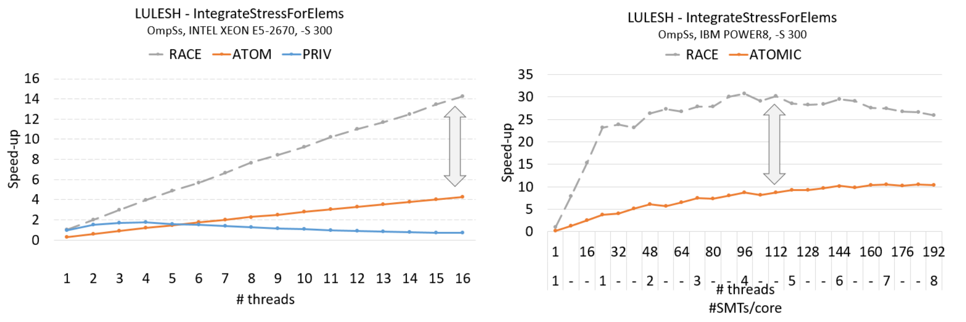


(a) Scalar-type reduction

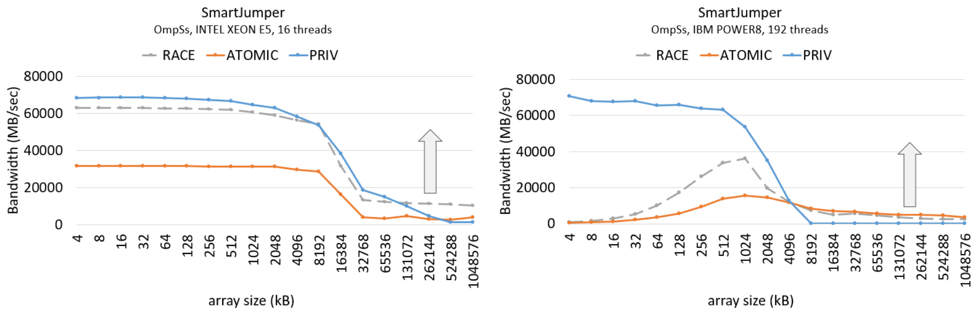


(b) Array-type reduction

Figure 2.2: Reduction support through replication redirects original memory accesses to the reduction variable r into thread-private data copies r_1 and r_2. This operation is preceded by a concurrent initialization and followed by a serial reduction of private copies into the original reduction variable.



(a) Scalability of the *IntegrateStressForElems* reduction kernel over different thread counts



(b) Scalability of the SmartJumper micro benchmark over different problem sizes

Figure 2.3: Concurrent executions of LULESH and SmartJumper on different systems show the degree at which atomic updates and privatization affect performance. While for LULESH, privatization results in significant performance degradation due large overheads, it is the technique of choice for SmartJumper and small data sets (2.3(a)).

Furthermore, privatization does not address the inherently low data locality of irregular accesses. Figure 2.2 shows a schematic representation of this support technique for scalar and array types.

In Figure 2.3 we show the performance impact of both, atomic updates and data replication applied to the LULESH and SmartJumper application. SmartJumper is a modern implementation of a random access benchmark that scatters memory updates over an array of variable sizes. Its irregular memory access pattern is shown in Figure 1.3(c) in Chapter 1. The measurements were taken on an Intel XEON E5 and IBM POWER8 system. The charts show achieved speedups expressed as speedup over serial execution time and fixed problem size for LULESH (2.3(a)) and achieved bandwidth with variable problem size for SmartJumper(2.3(b)). They show the performance impact of atomic updates and replication. Interestingly, while the use of replication produces large overheads due to the handling of private memory, this technique yields the best performance results for small array sizes as shown in the lower charts. In this case the overhead of private memory handling is small due to small array copies. Further, the use of thread-private copies, data sharing among processors can be avoided. This effect is visible on the POWER8 system for execution with 192 threads, where the use of replicas results in a significantly better performance. For bigger sizes, handling of large memory copies becomes the limiting factor. In these charts, *Race* refers to a hypothetical version with no applied technique that produces data races. Throughout our work we use this version as a reference for comparison.

Consequently, it is desirable to follow two design recommendations when implementing support techniques:

- Keep small data apart
- Remove atomic updates from the hot path of an application

Still the question remains if execution speeds can be improved for these algorithms and if similar or even better speedups can be achieved in comparison to *Race*. We discuss a technique that does so in Chapter 6 and present more results in Chapter 8.

2.2 Related Work

In this section we take a look at the most influential work on supporting reductions in software and hardware.

One commonly known technique to avoid the large memory overheads of privatization is called selective privatization, (*SelectPriv*). This approach inspects accesses by traversing the iteration space and marks conflicting elements in a hash table. During execution, called the executor phase, the hash map is consulted for each memory access. In case an entry is marked

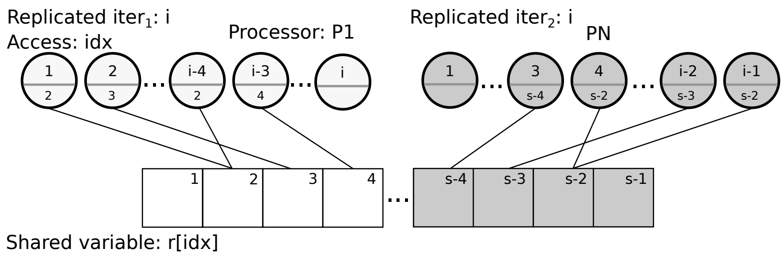


Figure 2.4: Schematic overview of *LocalWrite*, a techniques that implements an "owner computes" execution model by replicating the iteration space and selectively executing only those iterations that result in updates of owned memory.

as conflicting, the update is redirected to private memory. At the end of the computation, privatized data is reduced to global memory. In case of many conflicts, it converges to privatization. In our work we have implemented selective privatization as an additional feature to our techniques with access redirection. More information on this technique as well as experiences gained from its use are discussed in Chapters 7 and 8.

Another technique that avoids data races falls into the category of iteration reordering and is called *LocalWrite*. It implements an *"Owner computes"*-policy. This approach defines ownerships over regions of the reduction array and assigns iterations (loop indexes) to processors depending on the ownership of the resulting memory access location. This significantly improves data locality as each processor updates local data only. Implementations differ. One implementation builds on top of iteration space replication. In this case, each processor iterates over the entire iteration space but updates only those iterations that result in updates of owned memory. A schematic overview of LocalWrite with iteration space replication is shown in Figure 2.4. Other implementations rely on descriptors that mark relevant iterations and help to avoid iterating over unnecessary section of the iteration space. Unfortunately, in scenarios where many iterations update the same data causing many conflicts, execution is serialized. Similarly to SelectPriv, LocalWrite requires an inspector-executor execution model.

A particular implementation of a LocalWrite approach is called *Synch-Write* [10] [11]. The idea is that the iterations space is partitioned based on the range of accessed data. Such partitions can be processed in parallel to each other using synchronization to separate groups of conflicting iterations into different execution phases. While it is considered as a scalable scheme for near-linear or not non-overlapping access patterns, its execution performance is limited by underutilization of processing power in the opposite case.

A comparison of LocalWrite, SelectePriv and replication was conducted by Han and Tseng [3]. Their work discusses each approach as well as their implications on performance in respect to key algorithmic properties.

An automated decision framework based on algorithmic properties was presented by *Yu* and *Rauchwerger* [12]. In their work they identified a multitude of different static and dynamic properties of reduction kernels that allowed them to develop a decision tree. Using that tree, a runtime system could select a technique for a particular problem. This work led to the publication by *Yu*, *Dang* and *Rauchwerger* [13] where the decision making process was further formalized by introducing a pattern descriptor and a decision tree learning algorithm.

We think that automated decision making is beneficial for programmers as it lowers the requirement to know system or even algorithmic properties and follows the trend of abstraction. However, we direct our attention more towards investigating generic and extensible solution frameworks that allow programming model implementors to provide a portfolio of documented techniques instead. This approach favors programming transparency and leaves decision making to the programmer. In Chapter 7 we discuss this in more detail.

Figure 2.5 shows a landscape of algorithms and associated software techniques as discussed in this section. The picture shows a grouping that we propose based on the underlying strategy of direct access, iteration ordering or access redirection.

On hardware support for reductions, two main contributions were made that also reflect the general direction of thinking. Firstly, *Ahn*, *Erez* and *Dally* propose the addition of a functional unit that includes a local memory (a combine store) to either the memory interface of the processor or to

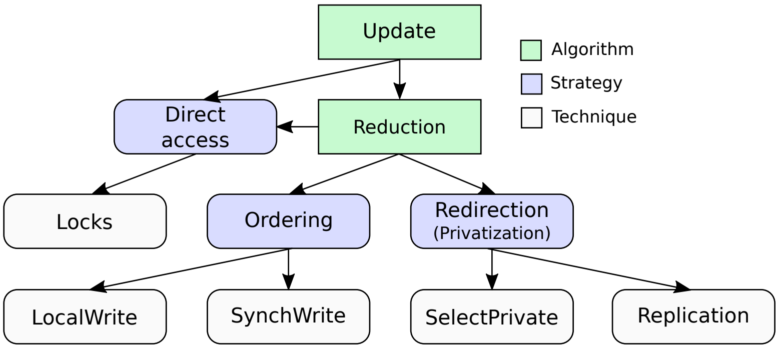


Figure 2.5: Landscape of algorithms, support strategies and techniques

the memory controller [14]. If access to an address corresponding to a memory location of an ongoing reduction is detected, a new entry in the local memory of the functional unit is created, initialized to the neutral element and immediately returned to the requesting processor. In the meantime, the functional unit issues a request to the memory subsystem. Once data arrives, it is combined with the current, temporal result located in the combine store by the functional unit itself. Address lookup is implemented via CAM (content addressed memory). The functional unit proposed in this work implements an adder only but the addition of more functional units is thinkable.

Garzarán et al. propose a similar approach but instead of using functional units with local memories, they propose the use of private cache lines [15]. Private cache lines hold temporal reduction data provided from the memory controller to the processor on request. Private cache lines are excluded from cache coherence and serve for the purpose of aggregating temporal, processor-private results. In case a cache line is evicted, the memory controller performs a load from memory, combines it with the temporal results and writes it back to memory.

We believe that both hardware assisted approaches are feasible and also of interest to us as they coincide with our future work of implementing a push semantic towards memory to avoid data round trips.

2.3 The OmpSs Programming Model

OmpSs is the representative task-parallel programming model used in this work for feature development and experimentation. Therefore in this section we give a quick introduction to this programming model.

OmpSs is a high-level, task-based, parallel programming model supporting SMPs, heterogeneous systems and clusters. It consists of a language specification, a source-to-source compiler for C, C++ and Fortran [16] and a runtime [17]. The language defines a set of directives that allow a descriptive expression of tasks. Further, OmpSs allows the programmer to annotate task parameters with an input or output semantic depending on access type of this parameter within that task. This information establishes a producer-consumer relationship between tasks, also called task dependency or data flow. Due to the internal representation of these relationships in a directed, acyclic graph (DAG) with nodes being tasks and edges representing relationships, this semantical information is also called directionality information of a parameter or directionality for short.

In OmpSs, a task is defined as

```
#pragma omp task in(list) out(list) [other_clauses]
structured-block
```

with *in* and *out* being the corresponding directionality clauses.

With this information the runtime is capable of automatic scheduling of tasks that maintains correctness of code while alleviating the programmer of implementing manual synchronization. While this is similar to tasking in the recent specification of OpenMP, the OmpSs runtime implements a different execution model. In OmpSs, every application starts with a predefined set of execution resources and an explicit parallel region does not exist. This view avoids the exposure of threading to the programmer as well as the requirement to handle an additional scope, the scope of a parallel region.

At compile time, the OmpSs compiler processes pragma annotations and generates an intermediate code file. This file includes both, user code as well as all required code for task generation, synchronization and error handling. In the final step of compilation, OmpSs invokes the native compiler to create a binary file. At runtime, the main thread progresses through code, creates tasks and stops at synchronization points (explicit or implicit *barriers*).

Task creation is composed of creation of that task object itself that carries all descriptive information and of handling of its dependencies. Once a task object has been created, the runtime inspects the dependency graph to determine the relationship to previously created tasks. If a dependency has been found, progression into deeper levels of the graph can be stopped and a representative node is added to the graph. In the opposite case, the task is placed into a ready queue.

2.3.1 Handling Reductions in Data-flow Graphs

Reductions, due to the read-modify-write sequence, represent *inout* dependencies. Naturally this type of dependency serializes execution. Concurrency can be restored by overriding this dependency between reduction tasks. This can achieved by defining reduction variables as *concurrent.* Since OmpSs implements this feature already, reduction support in OmpSs builds on top of this infrastructure.

Figure 2.6 shows a sample application that includes an initialization task, 4 commutative tasks and one additional task that frees memory. Its dependency graph is shown in Figure 2.7.

```
#define SIZE 1000
int main( int argc, char** argv )
{
  int sum=0, start=0,* array=(int *)calloc (sizeof (int), SIZE);

  #pragma omp task out(array, sum) shared(sum) label(init)
  {
    sum=0;
    for (int i = 0; i < SIZE; i++) array[i] = 1;
  }

  for(int j=0; j<4; j++)
  {
    #pragma omp task in(array) concurrent(sum)\
                              label( concurrent)
    {
      for (int i = start; i < start+SIZE/4; i++){
        #pragma omp atomic
        sum += array[i];
      }
    }
    start+=SIZE/4;
  }

  #pragma omp task in(array,sum) label(free)
  {
    printf("%i\n",sum);
    free(array);
  }

  #pragma omp taskwait
  return 1;
}
```

Figure 2.6: A complete application shows the use of different OmpSs pragmas to implement a parallel array-sum. Reduction support is based on the functionality implemented for the *concurrent* clause.

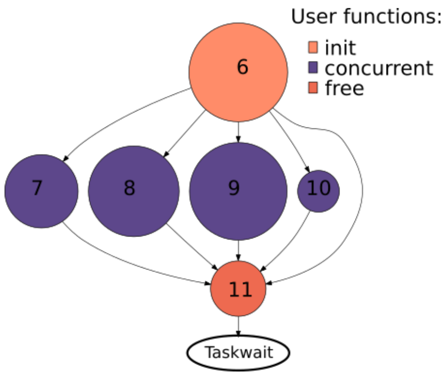


Figure 2.7: A dependency graph generated for the application shown in Figure 2.6, shows three different node types representing program methods and directed edges representing dependencies among them.

Chapter 3

Task-parallel Reductions in OmpSs and OpenMP

In this chapter we present an extension to the OmpSs and OpenMP *task* construct to add support for reductions in while-loops and general-recursive algorithms. Further we discuss implications on the OpenMP standard and present a prototype implementation in OmpSs. We explain the concepts of multi-level and nested-level domains as a solution to define the scope of a task-parallel reduction and also implement on-demand (dynamic) private memory allocation to minimize overheads associated to privatization. Benchmark results confirm applicability of this approach and scalability on current SMP systems.

3.1 Introduction to Task-parallel Reductions

Taking a broader look at usage patterns across applications reveals three common types of reductions: *for-loop* (bounded loop), *while-loop* (unbounded loop) and *recursive.* For-loop reductions enclose a reduction in a for-loop body. They are often used in scientific applications to update large arrays of simulation data in each simulation step (such as updating particle positions by a displacement corresponding to a time slice) or in numerical solvers where values are accumulated over a scalar to indicate convergence behavior and break conditions [5]. They are often referred to as array or scalar reductions.

For-loops represent the class of primitive-recursive algorithms where the iteration space is computable and where control structures of no greater generality are allowed. The iterative formulation of primitive-recursive functions is currently supported in OpenMP.

While-loop reductions represent another usage pattern and define the class of general-recursive functions. They appear in algorithms where the iteration space is unknown such as in graph search algorithms.

The last occurrence represents recursions. Recursive reductions can be found in backtracking algorithms used in combinatorial optimization. Even though one could argue that for each recursion an iterative formulation exists (either as a for-loop or a while-loop), recursions often allow very compact and readable formulations. Examples of a recursive and while-loop reductions are shown in Figure 1.4.

In this work we propose an extension to OmpSs by adding support for while-loop and recursive reductions through the *task reduction* directive. Formally, this extends the existing support for primitive-recursive, iterative algorithms by the class of general-recursive algorithms for both, iterative and recursive formulations. In terms of parallel programming, the proposed task reduction allows the expression of so called task-parallel reductions. Further we propose a compliant integration into OpenMP and present a prototype implementation based on OmpSs.

3.2 Specification for OmpSs and OpenMP

The idea to support task-parallel reductions builds on top of the conceptual framework introduced with explicit tasking in OpenMP. Since tasking allows to express concurrent while-loops and recursions, it represents a convenient mechanism to support task-parallel reductions as well. For its definition we use the current standard specification [1] as a baseline and add a set of rules describing data consistency and nesting. While this work is written with a certain formalism in mind, it does not represent a language specification.

3.2.1 Definition

The task reduction directive[1] is defined as:

```
#pragma omp task [clauses] reduction (identifier : list)
structured-block
```

The *reduction* clause in the task construct declares an asynchronous reduction task over a list of items. Each item is considered as if declared *shared* and for each item a private copy is assigned for each implicit task participating in the reduction. At implicit or explicit barriers or task synchronization, the original list item is updated with the values of the private copies by applying the combiner associated with the *reduction-identifier*. Consequently, the scope of a reduction over a list item begins at the first encounter of a reduction task and ends at an implicit or explicit barrier or task synchronization point. We call this region a *reduction domain*. Implications on synchronization in case of domain nesting is conforming to the OpenMP specification.

We would like to point out that the provided definition is generic and does not restrict the usage of task-parallel reductions to any particular enclosing construct. However, as in this case the scope of a task-parallel reduction is defined by both task synchronization as well as by barriers, its support would require to modify their current implementations. In particular they would need to check for outstanding private copies and reduce them. A solution to minimize the impact on unrelated programming constructs is to restrict the use of task-parallel reductions to the context of a *taskgroup*.

[1]Shown in C and C++ syntax

```
...
int red=0;
while(node->next) {
  #pragma omp task reduction (+:red)
  {
    red+=bar(node->value);
  }
  node=node->next;
}
#pragma omp taskwait
return red;
```

Figure 3.1: A concurrent reduction using a *taskwait* to ensure data consistency so a function would return a correct value of *red*

In the rest of this Chapter we discuss implications of this proposal on *taskwait* and *taskgroup* directives, reductions on data dependencies and nesting.

3.2.2 Reductions on taskwait

The *taskwait* construct specifies a wait on the completion of child tasks in the context of the current task and combines all privately allocated list items of all child tasks associated with the current reduction domain. A taskwait therefore represents the end of a domain scope. The previous example shown in Figure 2.1(a) can now be easily parallelized as shown in Figure 3.1.

3.2.3 Support in taskgroups

The *taskgroup* construct specifies a deep wait on all child tasks and their descendent tasks. After the end of the taskgroup construct, all enclosed reduction domains are ended and original list items are updated with the values of the private copies. Similarly to a taskwait construct, task-parallel reductions require to extend their role of task synchronization to actively perform a memory operation to restore consistency. Figure 3.2 shows an example where a reduction domain is ended implicitly at the end of a taskgroup construct.

```
...
int red=0;
#pragma omp taskgroup
{
  while(condition(){
     #pragma omp task reduction (+:red)
     red += foo();
  }
}
return red;
```

Figure 3.2: A concurrent reduction within the *taskgroup* performs a wait on all children and their descendant tasks (this is often referred to as "deep wait")

3.2.4 Reductions on data dependencies

Data-flow based task execution allows a streamline work scheduling that in certain cases results in higher hardware utilization with relatively small development effort. Task-parallel reductions can be easily integrated into this execution model but require the following assumption. A list item declared in the task reduction directive is considered as if declared *inout* by the *depend* clause. As this would effectively serialize the execution of reduction tasks due to the "inout" operation over the same variable, dependencies between reduction tasks of the same domain need to be overridden.

An example, where a reduction domain begins with the first occurrence of a participating task and is ended implicitly by a dependency introduced by a successor task, is shown is Figure 3.3. In this example the actual reduction of private copies can by overlapped by the asynchronous execution of *bar* which again might improve hardware utilization.

3.2.5 Nesting support

Nested task constructs typically occur in two cases. In the first, each task at each nesting level declares a reduction over the same variable. This is called multi-level reduction. In this case, a taskwait at each nesting level is not mandatory as long as a deep wait ensures proper synchronization later on. It is important to point out that only task synchronization that occurs at

```
...
int red=0;
for(int i=0; i<SIZE; i+=BLOCK){
  #pragma omp task shared(array) reduction (+:red)
  for(int j=i; j< i+BLOCK; ++j){
    red += array[j];
  }
}
#pragma omp task
bar();
#pragma omp task shared(red) depend( in:red)
printf("%i\n",red);
...
```

Figure 3.3: The reduction domain over the variable *red* is ended by a task dependency

the same nesting level at which a reduction scope was created (that is the nesting level that first encounters a reduction task for a list item), ends the scope and reduces private copies. Within the reduction domain, the value of the reduction variable is unspecified. An example for a multi-level domain reduction is shown in Figure 3.4.

In the second occurrence each nesting level reduces over a different reduction variable. This happens for example if a nested task performs a reduction on task-local data. In this case a taskwait at the end of each nesting level is required. We call this occurrence a nested-domain reduction. Figure 3.5 shows an example of an element-wise matrix summation, where inner tasks iterate over rows and compute partial results that are then reduced by outer tasks to compute the final value. Since in this example the nested domain is ended by the inner taskwait, accessing *red_local* returns a correct value.

3.2.5.1 General nesting support

The general support for nesting requires to inspect scenarios where a task-parallel reduction is nested within the *parallel* and *worksharing* constructs.

An example for such a scenario where a reduction is computed over a shared variable in a *parallel for* construct on one level and reduction tasks on the second level is shown in Figure 3.6. This example represents a multi-domain

```
...
int red = 0;
for(int i=0; i<SIZE; i+=BLOCK){
  #pragma omp task shared(array) reduction (+:red)
  for(int j=i; j< i+BLOCK; ++j){
    #pragma omp task shared(array) reduction (+:red)
    red += array[j] + bar(/*long_computation*/);
  }
  #pragma omp taskwait
  printf("Unspecified value of red:%i\n",red);
}
#pragma omp taskwait
...
```

Figure 3.4: A multi-level domain reduction is computed over the same variable by tasks participating at different nesting levels

```
...
int red = 0;
for(int i = 0; i < SIZE_Y; i++){
  #pragma omp task shared(array) reduction(+:red)
  {
    int red_local = 0;
    for(int j = 0; j < SIZE_X; j+=BLOCK_X) {
      #pragma omp task reduction(+:red_local)
      for (int k = j; k < j + BLOCK_X; ++k){
        red_local += array[i][k];
      }
    }
    #pragma omp taskwait
    printf("Correct value of red_local:%i\n",red_local);
    red += red_local;
  }
}
#pragma omp taskwait
...
```

Figure 3.5: Element-wise matrix sum implemented as a nested-domain reduction, where each dimension is processed in a different nesting level over a different variable

```
...
int red = 0;
#pragma omp parallel for shared(array) reduction (+:red)
for(int i=0; i<SIZE; i+=BLOCK){
  for(int j=i; j< i+BLOCK; ++j){
    #pragma omp task shared(array) reduction (+:red)
    red += array[j] + bar(/*long_computation*/);
  }
  #pragma omp taskwait
}
...
```

Figure 3.6: Nested task-parallel reduction in a worksharing construct performs a reduction over the shared variable *red*

reduction because even though both directives declare a reduction over the same variable (similarly to Figure 3.4), the inner reduction is performed on private copies that were created for each implicit task of the parallel region. In this case the outer reduction domain starts at the encounter of the first reduction task (implicit in this case) and ends at the implicit barrier at the end of the parallel region. The inner reduction domain starts on each thread with the encounter of the first explicit task and ends at the taskwait.

In case the reduction variable in the work-sharing construct would be declared *shared* instead, each implicit task would perform a reduction on the shared variable by its nested reduction tasks. Here the scopes of the inner reductions would end at implicit synchronization points within the implicit tasks and the runtime would need to make sure to update the shared variable atomically.

Currently OpenMP does not support nesting data-parallel and task-parallel reductions because of the following restriction.

- A list item that appears in a reduction clause of the innermost enclosing worksharing or parallel construct may not be accessed in an explicit task.

Adjusting this restriction as shown below, would add support for the general nesting support while maintaining the afford of discouraging programming errors.

```
int count = 0;
int nqueens(...){
  if (cond1(...))
    return 1;
  for(int row=0;row<n;row++){
    if (cond2(...)))
       #pragma omp task\
       reduction(+:count)
       count += nqueens(...);
  }
  #pragma omp taskwait
  return 0; //neutral element
}
```

(a)

```
int nqueens(...){
  if (cond1(...))
    return 1;
  int count = 0;
  for (int row=0;row<n;row++){
    if (cond2(...)))
       #pragma omp task\
       reduction(+:count)
       count += nqueens(...);
  }
  #pragma omp taskwait
  return count;
}
```

(b)

Figure 3.7: A concurrent implementation of N-Queens as a multi-level (a) and nested-level domain (b) reduction over the variable *count*

- A list item that appears in a reduction clause of the innermost enclosing worksharing or parallel construct may not be accessed in an explicit task unless it appears in its reduction clause.
- Nested reductions over the same list item must perform the same reduction operation.

If a general support of task-parallel reductions as discussed in this sections is desirable depends on its necessity. Currently task-parallel reductions enclosed in the taskgroup construct represent a satisfactory approach that reduces the impact of tasking on barriers.

Figure 3.7 shows two implementations of the n-Queens application from Chapter 1, that compute a reduction over a global (a) and local (b) variable.

3.3 Implementation in OmpSs

To evaluate requirements for compilers as well as for runtime support we implemented the presented proposal in the OmpSs programming model. In OmpSs, an application is launched as a single implicit task in an implicit parallel region over all available threads. Therefore neither the parallel

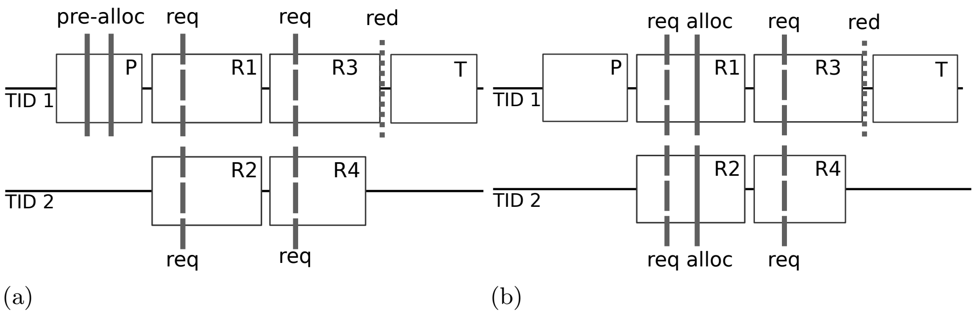


Figure 3.8: Static (a) and dynamic (b) allocation differs in when and where thread-private memory is requested (req), allocated (alloc) and reduced (red)

construct nor barriers are needed and memory consistency is ensured through data dependencies and task synchronization directives. Even though these differences exist, OmpSs is suited to serve as a reference implementation for the specific use-cases presented in section 3.2.

3.3.1 Runtime support

The runtime implementation is based on the idea of privatization. In order to avoid the need for mutual exclusive access to the reduction variable, a thread-private copy (TPRS) is created and used as a temporal reduction target. Since its creation, initialization and processing later on are expensive operations, it is important to maximize the life span and reuse of a TPRS.

Therefore we introduce a thread-team private reduction manager object that tracks privatized memories and assigns them to requesting tasks. Consequently all tasks that are executed on the same thread and belong to the same reduction domain always receive the same allocated thread-private memory. Once the domain ends, one of the participating threads reduces all corresponding TPRSs serially.

3.3.1.1 Allocation strategies and storage handling

To evaluate memory allocation we implemented two strategies called static and dynamic allocation.

Static allocation preallocates and initializes an array of thread-private reduction storages for all threads of a team (as defined by *omp_get_num_threads()*) at the moment when the first reduction task is created. This marks the beginning of a reduction domain. During execution, the runtime provides previously allocated TPRS objects according to a domain and thread identifier to requesting child tasks.

With dynamic allocation, memory is allocated and initialized on demand at task execution. Once a task requests a thread-private storage, the runtime performs an allocation, registers the storage in the reduction manager and returns a TPRS. An allocation is performed for each first execution of a reduction task of a domain on a participating thread. In this case a reduction domain begins at the execution of the first reduction task. This allocation strategy does not create any work for the encountering thread as the allocation is called at execution time of child tasks in parallel.

A schematic execution diagram of an application where a parent task *P* creates four reduction tasks *R1-4* running on two threads (*TID 1, 2*) is shown in Figure 3.8. In this scenario a following task *T* has an input dependency on the reduction variable and expects a correct value in memory at the moment of its execution. In OmpSs, finding the right point in time to reduce TPRSs is implemented via data dependencies. TPRSs are reduced in the moment when data dependencies are satisfied, or in other words, when the last task of a dependency domain has finished execution. In this example, reduction task *R3* is the task that satisfies the dependency requirements for task T and once completed it instructs the current thread *TID 1* to reduce all private storages of the reduction domain.

The advantage of static allocation is that it allows to allocate memory in a single call (to malloc for example) and its implementation is lock free once all TPRSs have been allocated. On the other hand, allocation is in the critical path and potentially can result in allocating unused storage in case not all threads participate in the computation. Further this approach does not adapt to changing numbers of participating threads.

Dynamic allocation allocates and initializes memory in parallel and avoids unnecessary allocation for busy threads that will not participate in the reduction computation. Since the number of registered TPRS storages changes over time, this implementation requires a lock in the global manager which can potentially introduce lock contention for fine grained tasks. This approach corresponds to the idea of dynamic parallelism where neither problem size nor thread counts are known.

3.3.1.2 Nesting support

In case of nesting, synchronization constructs might occur at any nesting level. In this case the runtime must be able identify storage locations that correspond to an ending domain.

For this purpose the reduction manager object implements a list of TPRSs and two maps that point to individual items in that list. One map uses task identifiers while the other one uses target addresses (pointers to the original reduction variable) as primary keys.

At the first execution of a task of new reduction domain, a new TPRS is allocated and initialized for the current thread and the parent work descriptor identifier as well as the address of the reduction variable are stored in the corresponding maps. Each successive task running on that thread and reducing over same variable will receive the same TPRS because of matching addresses. Once tasks finish, and the recursion starts to collapse, only those tasks that have a matching task identifier stored in the map are allowed to reduce TPRS storages. This corresponds exactly to those tasks that created a new reduction domain.

3.3.2 Compiler support

The goal of the Mercurium compiler [18] is to generate code transformations according to OmpSs annotations provided by the programmer. In case of encountering a task reduction, the compiler replaces all occurrences of the original reduction variable within the task by a reference to a previously requested thread-private reduction store, called TPRS. This transformation includes the following steps.

- Generate call to the runtime to obtain a TPRS. The runtime serves a TPRS corresponding to the current thread that is executing the task
- Replace all references to the original reduction variable within the task by a reference to the TPRS
- Apply the above transformation on the final task code block (code that is executed in the final task region).

The final task code block represents the original user-written code without runtime calls and is invoked when the task's final clause evaluates to true. The final clause in the task construct is typically used to set a cut-off value for task generation in order to control task granularity. In this way recursive functions can be stopped from task generation in order to avoid large runtime overheads.

In case of task reductions, multiple tasks can invoke the final code block that is performing a reduction over a global reduction variable in parallel. Consequently in order to avoid race conditions, accesses to the reduction variable within the final code block need to be redirected to the thread-private storage as well. Since requesting a TPRS is typically implemented as a runtime call, careful implementation is needed to minimize its impact on performance. Alternatively an additional final code block can be generated and invoked that accepts a TPRS pointer as an additional parameter. This would make the runtime call to request a TPRS obsolete.

Compiler transformations applied to Figure 3.4 are shown in Figure 3.9.

3.4 Evaluation

The evaluation of the presented runtime support is based on four application kernels that include while-loop and recursive reductions. The first two applications, n-Queens and Knight's tour, represent the satisfiability problem in numerical combinatorics. They compute in one case the maximum number of different configurations of n queens, in the other, the number of knight's paths covering all fields on a chess board of size n. These applications are implemented as recursive backtracking algorithms in two versions. In one version the reduction is performed over a task-local variable (nested-domain reduction) and in the other over a global variable (multi-level domain reduction). The

```
void outline_task1(struct ArgsTask1 args);
void outline_task2(struct ArgsTask2 args);
int foo() {
  ...
  for(int i=0; i<SIZE; i+=BLOCK)
    rt_create_task({array, i, &red}, &outline_task1);
  rt_taskwait();
  return red;
}
void outline_task1(struct ArgsTask1 args) {
  int *tp_red = rt_get_thread_storage(args.red);
  for(int j=i; j< args.i + BLOCK; ++j)
    rt_create_task({array, j, tp_red}, &outline_task2);
  rt_taskwait();
}
void outline_task2(struct ArgsTask2 args) {
  int *tp_tp_red = rt_get_thread_storage(args.tp_red);
  (*tp_tp_red) += args.array[args.j] + bar(/*long_computation*/);
}
```

Figure 3.9: Transformations applied by the compiler redirecting accesses to a thread-private reduction store

schematic concurrent code for n-Queens is shown in Figure 3.7. Execution traces obtained from the n-Queens application running on 16 threads are shown in Figure 3.10 and illustrate task execution and lifetimes of TPRSs for both implementations. In these executions, task granularity was set to an optimum by using the *final* clause in the task construct. As visible in the execution trace, a multi-level domain reduction allows high reuse rates of thread-private memory across nesting levels.

Max-height, another application, computes the longest path over a directed, unbalanced and acyclic graph. This application represents a while-loop reduction. Due to its frequent, irregular memory accesses, its scalability is limited by memory bandwidth when running on 16 processor cores. The Powerset benchmark computes the number of all possible sets over a given number of elements. This application is implemented recursively where unlike the n-Queens application, each recursive branch is of the same length.

Figure 3.11 shows application scalability for the aforementioned applications implemented as nested-domain (a) and multi-level domain (b) reduction as

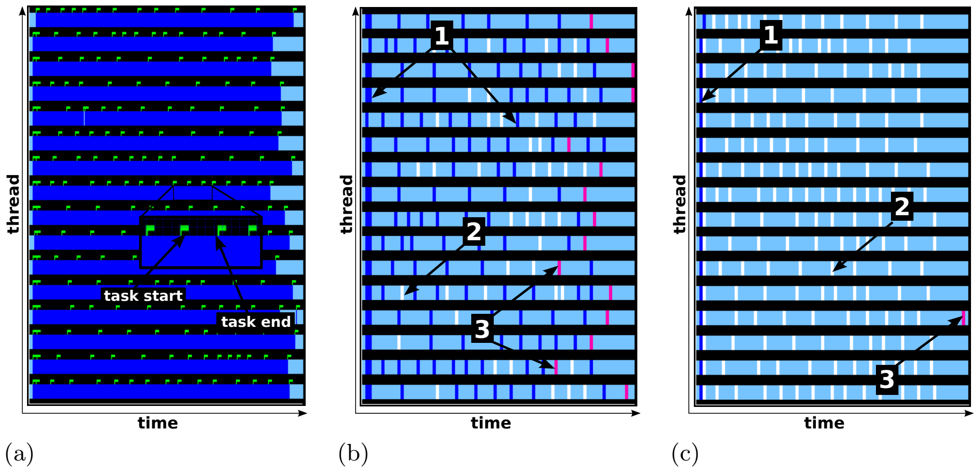
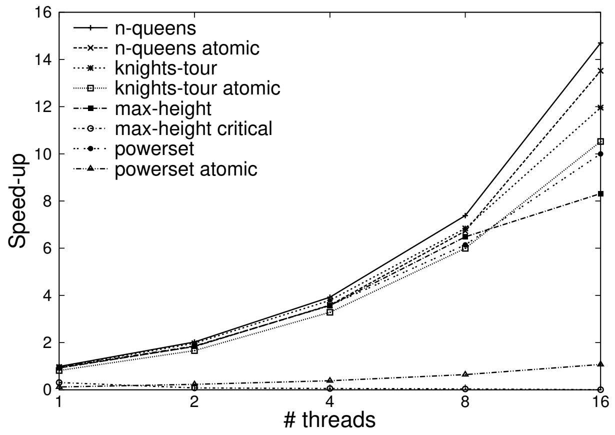


Figure 3.10: An execution trace for the n-Queens application (n=15 and creating tasks in two nesting levels) shows tasks (a) and allocation (1), reuse (2) and reduction (3) of thread-private reduction storages over a task-local (b) and global variable (c) as shown in Figure 3.7

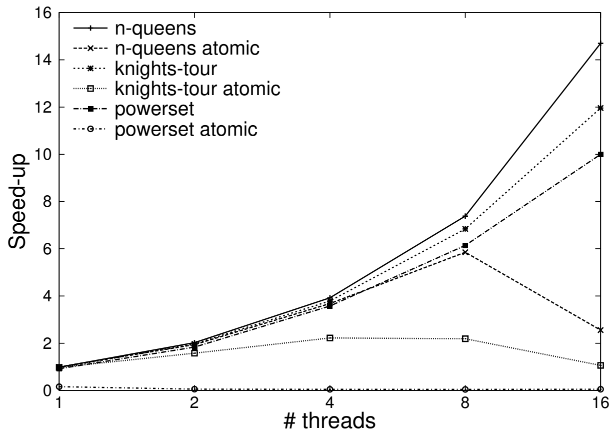
well as their implementations with tasking and atomic updates (using built-in atomics where possible). For each benchmark we have selected the best task granularity and the following problem sizes: n-Queens with n=15, Knight's tour with board size 5x5; Max-height with a graph height 15, 7 edges per node and Powerset over 32 items. Results show that all benchmarks benefit from reduction support, especially in cases where lock contention and atomic updates degrade performance.

While all applications were executed with both allocation strategies, they did not exhibit significant performance differences. Allocation and initialization strategies are relevant especially in case of user-defined reductions over larger data types and array reductions where the cost of memory allocation and initialization becomes expensive. As a more detailed analysis exceeds the scope of this chapter, we defer it to future work.

Inspecting overheads of the current implementation for both allocation strategies revealed that the additional time introduced by runtime calls specific to reduction support did not exceed 1% of total time spent in the runtime regardless of granularity or problem size. We expect that this behavior will change in case of user-defined and array-type reductions.



(a)



(b)

Figure 3.11: Application speed-up over serial execution implemented with (a) nested-domain and (b) multi-level domain reductions as well as with regular tasking using atomics

3.4.1 Environment

All benchmark results presented in this work were obtained from the MareNostrum 3 supercomputer located at the Barcelona Supercomputing Center. Each system node contains two 8-core Intel Xeon E5-2670 CPUs running at 2.6 GHz with 20MB L3 cache and 32GB of main memory. Applications were compiled using Mercurium compiler v1.99.1 and GCC v4.8.2 back-end/native compiler with -O3. The runtime is based on the Nanos++ RTL v0.7a.

3.5 Related Work

OpenMP has allowed concurrent reductions in work-sharing constructs since its first specification (OpenMP 1.0) in 1997. The supported clauses allow the use of a reduction operator and a list of scalar, shared locations. During parallel execution, the runtime creates private copies for each list item and thread in the team. The result variable is initialized with the identity value according to the operator that is declared in the clause. In successive versions of the OpenMP specification, additional features have been added to the standard. These include min- and max-operators for C/C++ in version OpenMP 3.1, extended reduction support to Fortran Allocatable Arrays (OpenMP 3.0) and User Defined Reduction (OpenMP 4.0). Aside from small incremental updates, the OpenMP specification has never allowed OpenMP tasking in reductions. In fact it explicitly forbids the use of reduction symbols in combination with tasks: "A list item that appears in a reduction clause of the innermost enclosing work-sharing or parallel construct may not be accessed in an explicit task." [1] This restriction reduces flexibilities achieved by dynamic parallelism in many algorithms.

Other programming models, such as the Cilk++ [19], introduce different types of linguistic mechanisms, so called hyperobjects [20], that are coordinating local views of the same variable. Based on a `cilk_spawn` mechanism that starts parallel execution, the parent creates a private copy of the original view initialized with the identity value, while the child receives the original view of the symbol. These two views join just before synchronization occurs (`cilk_sync`). In the first step, the child view is updated with the value of the parent view according to a reducer function. Then the parent view is discarded, and replaced by the view of the child. Unlike lazy-reduction implemented in OmpSs that is able to reuse storage across nesting levels, in this case an allocation, reduction and deallocation always occur.

X10 [21], another popular programming model, introduces a phaser-accumulator [22, 23] construct for dynamic parallelism. A X10-phaser is a coordination construct allowing to unify point-to-point synchronization among different X10-tasks (activities). The phaser-accumulators support two logical operations. It sends a value for accumulation that has been produced in the current phase or it receives the accumulated value from the previous phase. This implementation entirely eliminates race conditions through the accumulator object that handles read and write accesses. This encapsulation of functionality allows to define implementation strategies that differ as to when the reduction itself is performed. That is either when data is supplied or when a synchronization point is reached. In this respect this implementation is comparable to OmpSs.

3.6 Conclusions

In this chapter we presented an extension to OpenMP tasking to support reductions in while-loops and general-recursive functions. It turned out that the OpenMP taskgroup is suited to support task-parallel reductions as it minimizes implications on unrelated constructs. A general support in OpenMP is possible but requires further analysis of deep task synchronization and of implementation and performance implications on barriers. This effort should be made in the future if applications exist that render the taskgroup construct insufficient. The presented runtime implementation offers two different allocation strategies and maximizes storage reuse. Dynamic allocation follows the idea of dynamic parallelism where neither the amount of work nor the number of participating threads are known beforehand. Results show that task granularity is important and in case of recursive algorithms can be efficiently controlled by the final clause. In the case of while-loops, task granularity needs to be taken into account by the programmer through appropriate application design. Performance results obtained on a MareNostrum 3 system node show a near-linear speed-up for test cases with optimal granularity.

Chapter 4

Towards Task-parallel Reductions in the OpenMP Specification

The introduction of OpenMP tasking enabled new parallelization opportunities for irregular algorithms. Unfortunately the tasking model does not easily allow the expression of concurrent reductions, which limits the general applicability of the programming model for such algorithms. In this chapter, we refine our previous work on task-parallel reductions with the goal to produce a compliant version to the current OpenMP specification. In particular we restrict the possible use of task-parallel reductions to task groups and introduce the *in_reduction* clause. Further we explore issues for programmers and software vendors regarding programming transparency as well as the impact on the current standard with respect to nesting, untied task support and task data dependencies. Our performance evaluation demonstrates comparable results to hand-coded task reductions.

4.1 Introduction to Task-parallel Reductions in OpenMP

Migrating applications to multi-core and many-core architectures is a challenging but necessary step to achieve scalable performance on modern systems. Thus, parallel programming models such as *OpenMP* [1] have gained popularity through concepts and tools to introduce portable concurrency in a broad range of algorithms with relatively little programming effort. This work follows the line of thought and proposes the addition of task-parallel reductions to OpenMP to extend support to a wider class of algorithms. For-loops have a constant iteration space and OpenMP supports their concurrent execution through worksharing constructs. Unlike for-loops, the iteration space of while-loops and recursions is dynamic, which prohibits an efficient use of worksharing constructs. OpenMP 3.0 added support for these irregular algorithms through the `task` directive. In this formulation, loop iterations and recursive calls create task instances of the enclosed code, typically the loop body.

While for-loops and while-loops can be efficiently parallelized through worksharing constructs or tasks, reductions within them require special attention. A closer look reveals that the reduction operation represents a read-modify-write sequence that is not atomic so that its parallel execution introduces data races.

Figure 4.1 shows while-loop reductions over a linked list that avoid data races by introducing locks or by applying techniques like thread-privatization. Programming model support would eliminate the required boilerplate code. Even though manual implementations are viable solutions, they are error-prone and require the programmer to select a specific implementation, which may be inefficient on a given architecture or incur unnecessary memory overheads.

OpenMP needs a solution that supports task reductions and minimizes the effect on unrelated constructs. It should comprehensively define the scope of the reduction and a data context for the private reduction variable.

```
float var = 0;

   ...

   while ( node ) {
      var += node->value;
      node = node->next;
   }

   ...



...
```

(a) Original code (serial version)

```
float var = 0;
#pragma omp parallel
{
 #pragma omp single
 while ( node ) {
 #pragma omp task \
           firstprivate(node)
  {
    #pragma omp atomic
    var += node->value;
  }
 node = node->next;
 }
}
```

(b) Parallel with atomics

```

float var = 0;
float part[nthreads] = { 0 };

#pragma omp parallel \
            reduction(+:var)
{
 #pragma omp single
 {
  while ( node ) {
    #pragma omp task \
          firstprivate(node)
    {
     part[thread_id] +=
                node->value;
    }
    node = node->next;
  }
 }
 var += part[thread_id];
}
```

(c) Parallel with manual privatization

```
float var = 0;
float part = 0;
#pragma omp threadprivate\
                    (part)
#pragma omp parallel \
            reduction(+:var)
{
 #pragma omp single
 {
  while ( node ) {
    #pragma omp task \
          firstprivate(node)
    {
     part +=
            node->value;
    }
    node = node->next;
  }
 }
    var += part;
}
```

(d) Parallel with thread-privatization

Figure 4.1: Different versions of a while-loop reduction over a linked list

4.2 Discussion of Concerns

We propose to extend the *taskgroup* and *task* constructs to support task reductions. Prior work identified *taskgroup* construct as a possible scope of the reduction [24]. We prefer this choice since it does not affect other OpenMP mechanisms (e.g., barriers) and the taskgroup structured block defines a clear reduction scope.

We extend the *taskgroup* and *task* construct with the clauses *reduction* and *in_reduction* respectively. The *in_reduction* clause declares a task as a participant in the computation of *var* that was previously declared in an enclosing taskgroup *reduction* clause with the same *reduction-identifier*. We deliberately use the *in_reduction* clause instead of reusing the *reduction* clause in order to stress the differences in behavior to the programmer. The *reduction* clause in the *taskgroup* construct follows its current specification for other constructs. Alternatively, the *in_reduction* clause on a task construct defines an access pattern (an update operation) to one of those copies. Figure 4.2(a) illustrates our proposal for the previous example.

4.2.1 Updates of a reduction variable outside a reduction context

Programmers must consider that an update of the original reduction variable occurs just after the *taskgroup* region and that accesses to that outside of the taskgroup may create a race condition. Figure 4.2(b) shows code that updates the reduction variable both inside and outside a taskgroup reduction. The task created in line 8 can be executed concurrently with the taskgroup reduction update occurring at the end of the taskgroup created in lines $11 - 12$. This situation may also occur when multiple taskgroup reductions are working with the same variable simultaneously. The programmer must provide proper synchronization to avoid this situation. This requirement is analogous to existing restrictions on reductions:

> To avoid race conditions, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the reduction computation (line 20, p. 170 [1]).

```
float var = 0;

#pragma omp parallel
{
  #pragma omp single
  #pragma omp taskgroup \
          reduction(+:var)
  while ( node ) {
    #pragma omp task \
            firstprivate(node)\
            in_reduction(+:var)
    {
      var += node->value;
    }
    node = node->next;
  }
}




...
```

(a) While-loop reduction (tentative)

```
float var = 0;

#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    var++;

    #pragma omp taskgroup \
            reduction(+:var)
    while ( node ) {
      #pragma omp task \
              firstprivate(node)\
              in_reduction(+:var)
      {
        var += node->value;
      }
      node = node->next;
    }
  }
}
```

(b) While-loop reduction (race condition)

Figure 4.2: Examples of our proposal

4.2.2 Over-specifying the reduction identifier

The declaration of the reduction identifier in the *in_reduction* clause could be inferred from the *taskgroup* context and thus could be omitted to minimize the potential for programming errors. However, vendor feedback indicates that omitting the identifier could limit compiler optimizations, or at least introduce some additional overhead (i.e., registering the reduction inside the runtime) to perform these optimizations. OpenMP vendors may use the identifier to combine a local-copy of a reduction variable with the original/thread-copy (depending on the implementation approach), which specification of the identifier in the *in_reduction* clause would facilitate. Thus, we choose to require it.

4.2.3 Supporting untied tasks

Untied tasks can be suspended at a task scheduling point and later resumed on a different thread. Without proper handling, a task might resume execution on a different thread but still continue using the thread-private copy of the thread that started its execution, which could create a race condition. Tied tasks do not encounter this issue since they execute entirely on one thread even if they are suspended at some point. Thus, they can safely use that thread's copy as they will not be suspended while accessing it.

Several solutions could support untied reduction tasks. First, an implementation could not migrate any task (e.g., treat it as tied) if it is involved in a reduction even though it is declared as *untied*. This approach is simple but eliminates the potential benefit of untied task migration.

Alternatively, an implementation could introduce an additional local variable for each untied reduction task. This task-local variable must be initialized to the identity. A reference to the local variable would replace all references to the reduction variable inside the untied task. Finally, at the end of the task, the partial result stored in the task-local variable would be combined with the thread-private copy of the thread that finalizes the task. This approach supports tasks that migrate among threads at the cost of an additional task-local copy that must be initialized and an additional partial reduction per untied task.

Finally, the compiler could generate a request for the thread-private copy after each possible task scheduling point, thus supporting the use of the thread-private copy. The reduction task would then always access the thread-private copy of the thread that is executing it. This approach supports tasks that migrate among threads at the cost of repeatedly obtaining the thread-private location.

We recommend that the following be implementation-defined:

- Whether untied tasks involved in reductions can migrate;
- The number of private copies that are created for a task reduction.

The number of private copies could be defined as the number of tasks that participate in the reduction. Our recommendation thus allows an implementation to choose any of the above solutions (or a hybrid of them). Untied tasks could migrate and the number of private copies could be anything between the number of threads to the number of tasks.

4.2.3.1 Evaluating support for untied tasks

We use two benchmarks to evaluate the choice of supporting untied tasks by not migrating them or by introducing a new local copy per task. The first performs a reduction over a scalar. The performance of both versions is equivalent since the extra overhead introduced in the task-local approach is small in scalar reductions and the benchmark is well-tuned to obtain good performance using tasks so the extra overhead of the task-local version is insignificant compared to the task granularity.

Our second benchmark, Array Sum UDR (since it has a *User Defined Reduction*) reduces an array of structs to a unique struct. This struct has a static array of TS integers. The UDR's initializer sets every element of the struct to zero and its combiner adds the values of the two arrays. We choose this benchmark since it increases the cost to allocate and to initialize the extra copy and to perform its associated reduction.

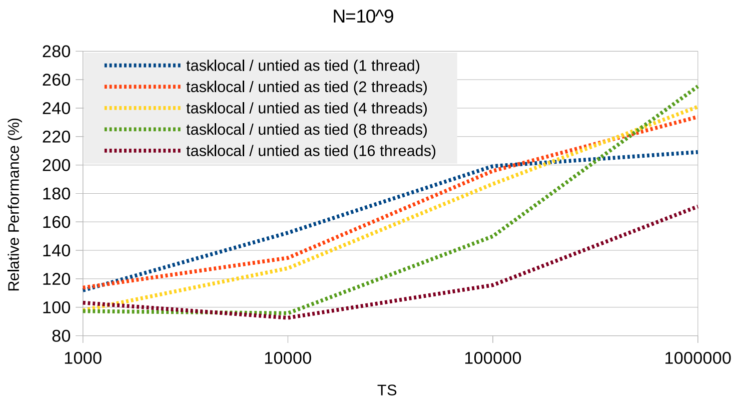


Figure 4.3: Array Sum UDR benchmark results

Figure 4.3 shows the relative performance of the task-local version compared against the untied-as-tied version, with different number of threads and fixing the total number of integers to $N = 10^9$. The relative performance is computed dividing the execution time of an approach by the execution time of another. Taking a closer look at the execution with a single thread for instance reveals that the introduction of a task-local variable doubles the execution time as initialization and reduction of this additional variable is required. Dividing this execution time by the execution time of the untied-as-tied version and normalizing the number to 100 results in a 200% speed-up of the latter for large problem sizes.

In general, the overhead of the task-local version increases with TS, the size of the static array. The differences among the different relative performances require further analysis which we defer to future work. Thus, the task-local approach is reasonable for scalar reductions but may incur excessive overhead for array reductions or UDRs.

```
int a = 0;
#pragma omp taskgroup \
                reduction(+:a)
{
    ...
    int b = 0;
    #pragma omp taskgroup \
                reduction(+:b)
    {
        ...
    }
    ...
    a += b;
}
```

(a) Nesting over two different variables

```
int a = 0;
#pragma omp taskgroup \
                reduction(+:a)
{
    ...

    #pragma omp taskgroup \
                reduction(+:a)
    {
        ...
    }
    ...

}
```

(b) Nesting over the same variable

Figure 4.4: Nested taskgroup reduction scenarios

4.2.4 Supporting nested taskgroups

Nested taskgroup reductions can be defined either over different list items or the same ones, as Figure 4.4 shows. If the nested taskgroup defines a reduction over a different list item (Figure 4.4(a)), the runtime registers a new reduction that is independent of the ongoing outermost taskgroup reduction. Thus, the runtime creates a new set of thread-private copies to compute the reduction.

Two alternatives exist if the nested taskgroup reduction is over the same list item (Figure 4.4(b)). The first uses the same approach as when the list item is different: register a new reduction. The second alternative reuses the same set of private copies for both reductions. With this approach, we cannot reduce the private copies at the end of the nested taskgroup reductions: the final reduction must be computed at the end of the outer *taskgroup* region, counter to current reductions semantics that compute the reduction at the end of the construct that has the reduction clause.

4.2.5 Cancellation, dependencies and merged tasks

Cancellation implies the value of the reduction variable is unspecified since we cannot guarantee how far the computation of the reduction has progressed. The programmer must anticipate this behavior.

The specification of a dependency (using the task *depend* clause) over a reduction variable might introduce a conceptually misleading situation. The programmer might intend a dependency over the original variable or the private copy in the data context of the taskgroup reduction. We could explicitly restrict the use of the *in_reduction* clause and depend clause over the same variable. However the current OpenMP specification does not restrict similar cases. A dependency over a private variable produces a similar situation where the OpenMP specification does not provide clarification about the interaction between data-sharing attributes and dependencies.

A merged task that participates in a reduction does not have a data environment. Thus, it must use the parent's data environment that includes the private copy of the reduction variable. Since the parent environment for a reduction task can only be either a taskgroup reduction or another reduction task environment, the use of the corresponding private copy[1] in the parent region is always guaranteed. Thus, this case also does not require additional specification.

4.3 Specification for OpenMP

This section describes the syntax of our proposal. We update the syntax of the taskgroup construct to:

`#pragma omp taskgroup` *[clause[[,] clause]...] new-line*
structured-block

where clause is:

`reduction(`*reduction-identifier: list*`)`

We also modify the *reduction* clause description to cover taskgroup regions. Once the scope of a reduction is defined, we must identify tasks within the taskgroup that participate in the computation. Thus, we extend the clauses allowed on a task construct to include:

`in_reduction(`*reduction-identifier* `:` *list*`)`

We add a section for the *in_reduction* clause and modify the description of the *reduction* clause to specify the semantics of references to the list items that we discussed in the previous section. The section on the *in_reduction* clause includes this restriction:

- The task to which the *in_reduction* clause is applied on a list-item must be closely nested in a *taskgroup* region to which a *reduction* clause is applied on the same list-item.

[1]This case may involve multiple private copies due to support for untied tasks.

4.4 Evaluation

This section compares the performance of our prototype implementation of the proposed taskgroup reduction with the manual implementation as shown in Figure 4.1(c).

4.4.1 System environment

We obtained our results on MareNostrum III and the Knight system located at the Barcelona Supercomputing Center. Each Marenostrum III node contains two 8-core Intel Xeon E5-2670 CPUs running at 2.6 GHz with 20MB L3 cache and 32GB of main memory organized as two NUMA nodes. Each Knight node includes an Intel Xeon Phi coprocessor with C0 silicon and board version C0PRQ-7120 (61 cores at 1238095 Khz, 16 GB of GDDR Memory at 5.5 GT/sec, 300W TDP), driver v3.4-1, MPSS v3.4 and flash v2.1.02.0390).

Applications on Marenostrum and Knight were compiled using the Mercurium source-to-source compiler v1.99.8[2] (using GCC v4.7.2 and Intel$^{(R)}$ C Compiler 15.0.2 as the back-end/native compiler respectively). In both cases the compiler optimization level was `-O3`, and the parallel runtime used in all experiments was based on the Nanos++ RTL v0.9a[3].

4.4.2 Benchmark descriptions

`Array Sum:` This algorithm takes a single array of N integers as an operand and computes the sum of its elements. We create a task for each TS elements.

`Dot Product:` The dot product algorithm is a simple operation on two vector operands of N elements. The result is the sum of the products of their components. We create a task for each TS elements.

`NQueens:` This application computes the number of placements of N chess queens on a $N \times N$ chessboard such that none of them can attack any other. This implementation uses a Branch and Bound algorithm following a recursive

[2]mcxx 1.99.8 (git 538d492)
[3]nanox 0.9a (git master 10f6134)

pattern, taskified and using the final clause to control task granularity.

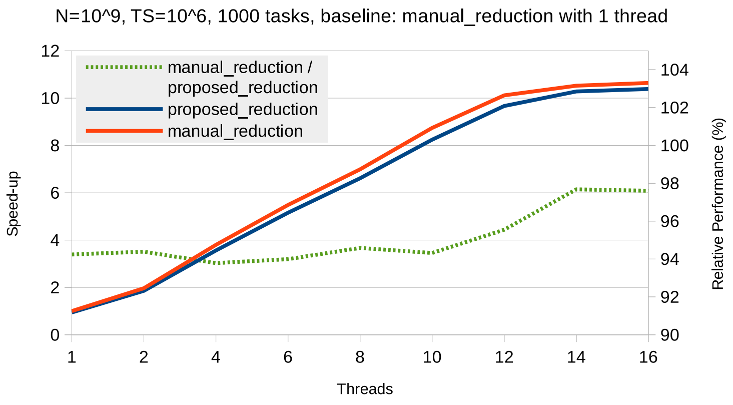
`Unbalanced Tree Search (UTS):` This benchmark computes the number of nodes in an implicitly defined unbalanced tree [25]. The program begins with a single tree node and an initial seed that is used to generate a sequence of pseudo-random numbers. For each node, the next value in the sequence is used to sample a parameterized probablity distribution to determine the number of children for a given node. This algorithm creates an unpredictably unbalanced workload that makes the use of a cut-off value in the final clause difficult.

4.4.3 Performance results on Intel Xeon processors

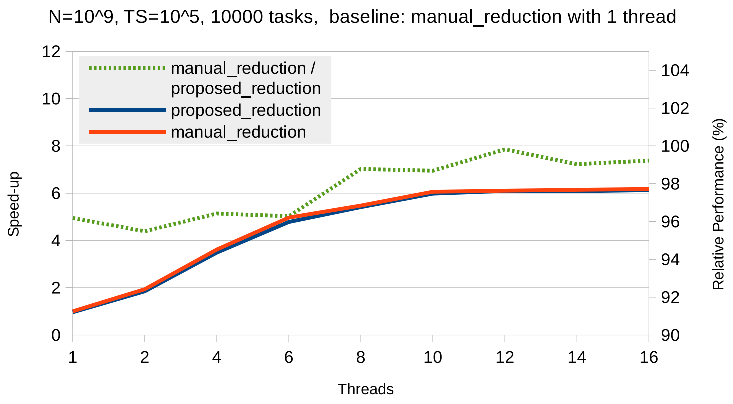
In this section we evaluate the performance of our proposal against the performance of manual versions of the benchmarks on Intel Xeon processors.

Figure 4.5 shows the performance results of the Array Sum and Dot Product benchmarks. Both benchmarks exhibit similar behavior in which performance drops levels off with higher thread counts. In this case, scalability is limited by memory bandwidth. In Array Sum, bandwidth saturation starts with 12 threads (with a 10x speed-up), while for Dot Product this effect becomes visible with 6 threads (reaching a speedup of 5x). These two different phases (scale and saturate) have a counterpart in the relative performance (the green dashed line in the figure). For all thread counts with Array Sum, the performance reaches at least 94% of the performance of the manual version. For larger thread counts, the differences between the implementations become smaller because task execution time shifts towards the computation as the algorithm saturates the memory bandwidth and reduces the importance of reduction performance. For the Dot Product benchmark, the relative speedup is between 95% and 100%. For both benchmarks, the gains in maintainability and portability easily compensate for the slight performance drop that occurs due to runtime handling of the reduction support.

Figure 4.6 shows the results for the NQueens benchmark. For this application we have implemented two versions: one that reduces over a global variable (subfigure a) and another that reduces over a local variable (subfigure b). We explore these two versions primarily because the global version only registers one reduction in the whole program while the local version registers a new re-

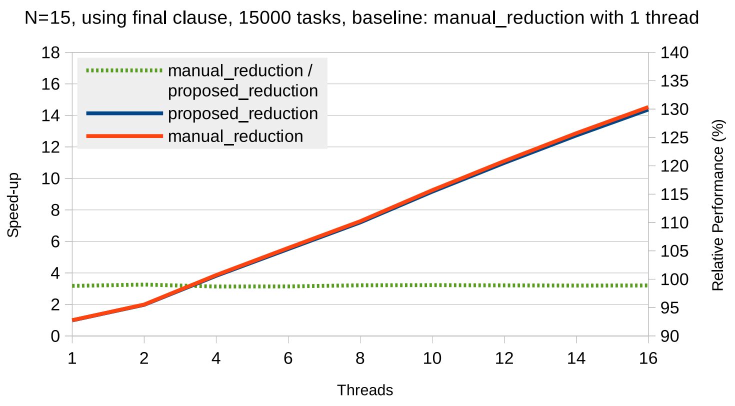


(a) Array Sum

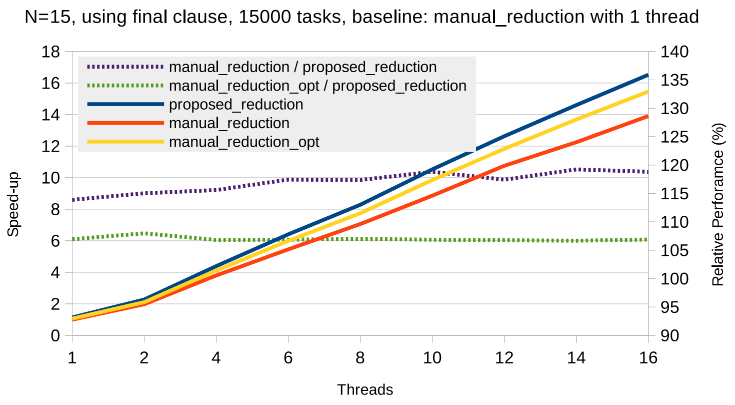


(b) Dot Product

Figure 4.5: Array Sum and Dot Product benchmarks results



(a) NQueens Global



(b) NQueens Local

Figure 4.6: NQueens benchmark results

duction at each recursive level. When reducing over a global variable, speedup is essentially linear and relative performance is close to 100%. When the reduction is performed over a local variable, we compare our proposal against two different manual versions. The first one is the regular transformation presented previously whereas the second version optimizes the code when in a final task. The problem with the regular transformation is that we are still allocating, initializing and reducing an array of $NUM\_THREADS$ elements even if we are going to use just one element. Thus, the optimized versions makes use of the *omp_in_final()* runtime service to avoid this extra overhead. Despite comparing our proposal against the manual optimized version, the scalability and the relative performance of our version is still better.

Figure 4.7 shows the results of executing the UTS benchmark with configurations that vary the number of created tasks from 50k to 1M tasks. All configurations achieve essentially linear speedup (subfigure a), and relative performance is between 96% and 99% for programmability issues again more than compensate.

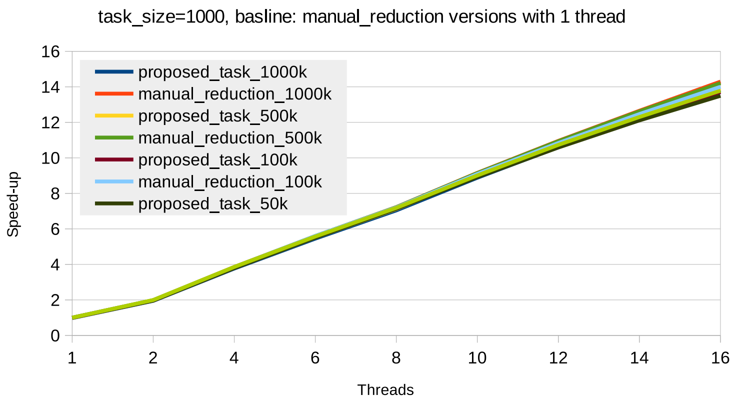
4.4.4 Performance results on Intel Xeon Phi coprocessors

In this section we evaluate the performance of our proposal against the performance of manual versions on a Intel Xeon Phi coprocessor.

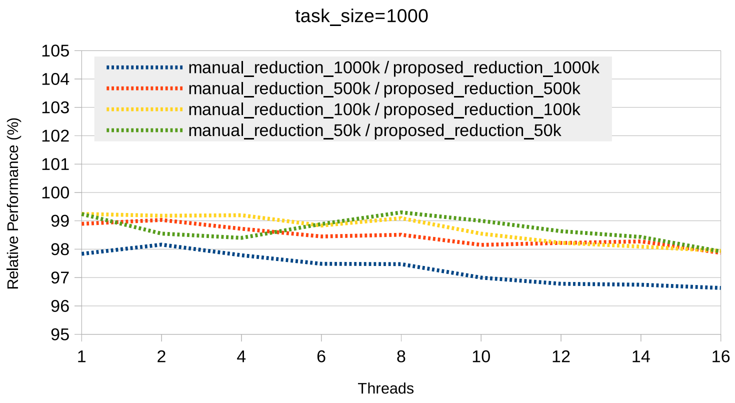
Figure 4.8 shows the results of the NQueens benchmark on the Xeon Phi. For the global version of the NQueens, the scalability and the relative performance between our approach and the manual version are identical. For the local version, the scalability and the relative performance of our proposal is equivalent to the manual optimized version and far better than the nonoptimized one.

4.5 Conclusions

In this chapter we have presented a proposal to support task-parallel reductions in OpenMP that extends the *taskgroup* and *task* constructs with *reduction* and *in_reduction* clauses. We consider that the *taskgroup* construct provides a convenient data environment for reductions and that the scope of the reduction is clearly defined by the deep synchronization at the end of the *taskgroup* region. The *in_reduction* clause for the task construct associates

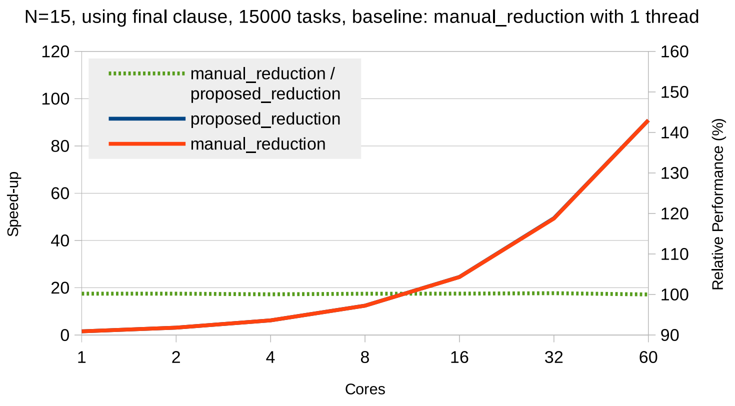


(a) UTS, Scalability

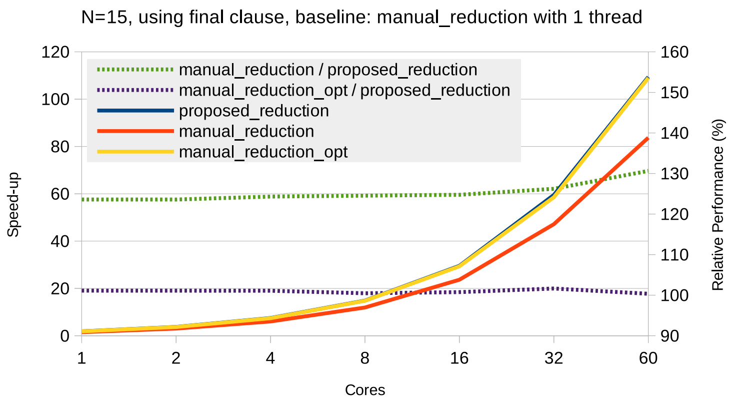


(b) UTS, Relative Performance (%)

Figure 4.7: Unbalance Tree Search benchmark results



(a) NQueens Global



(b) NQueens Local

Figure 4.8: NQueens benchmark results on Xeon Phi

tasks with a reduction previously declared in a *taskgroup* construct. This approach does not impact barriers or other task synchronization constructs. We explored implementation options to support nested taskgroups and untied tasks, which demonstrate that implementors can chose among a range of implementations and optimizations. Our performance results demonstrate that the approach incurs little overhead compared to manual versions and may even provide performance benefits in some cases like recursive applications.

Most importantly, it significantly reduces boilerplate code that programmers must currently use to implement reductions manually. This work represents the foundation for our specification draft presented to the OpenMP review board.

Chapter 5

Supporting Irregular Array-type Reductions on Distributed Memory Systems with CachedPrivate

Supporting irregular array-type reductions on distributed memory systems is a challenging endeavor due to frequent fine-grained accesses to high-latency, remote memory. A solution is needed that can maximize local and temporal data reuse and allow overlapped computation and communication. In this chapter we present an approach called *CachedPrivate* that fulfills these requirements by redirecting memory accesses into thread-private software caches with eviction buffers of variable size. Results confirm scalability for up to 32 12-core cluster nodes.

5.1 Introduction to CachedPrivate

In this work we propose a software-based approach, called *CachedPrivate*, that combines high programmability and an efficient parallel execution of reductions on distributed memory systems. These systems are typically characterized by high latency and low bandwith remote memory accesses which makes them well suited to evaluate our approach. To express a reduction, the user simply annotates an OmpSs program with an additional pragma directive. The underlying runtime takes care of its parallel execution. At its core, a software cache is used to privatize reduction data and to ensure scalability through improved data reuse (locality) and overlapped computation and communication that occurs on cache line evictions. The process of software caching is transparent to the programmer.

For practical evaluation we implemented the approach in OmpSs [26]. OmpSs is a task-based, parallel programming model that allows programmers to write parallel applications that seamlessly execute on shared- and distributed-memory systems (SMPs and clusters) and on heterogeneous systems (such as GPGPUs). The presented approach is not restricted to OmpSs but is generic and applicable to *OpenMP* [27, 28] as well as other parallel programming models.

The contribution of this work consists of:

- *CachedPrivate* – an approach based on software caching for scalable runtime support for irregular reductions
- language support for tasking and work-sharing constructs in C-like languages
- and performance evaluation

5.1.1 Occurrences and motivation

Reductions can occur on scalar and array types. For example a vector dot-product represents a reduction on a scalar whereas a histogram, as shown in Figure 5.1, represents a reduction on a vector. To compute a histogram a loop iterates over an input array of keys and updates the number of occurrences

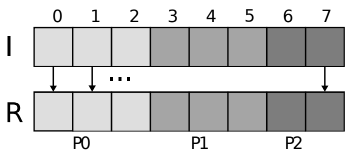
```
void Histo(TYPE* keys, TYPE* histo, int K) {
  for(int i = 0; i < K; i++)
  {
    TYPE key = keys[i];
     histo[key]++;
  }
}
```

Figure 5.1: Sequential histogram code

of each key. In this case *histo* is an array of reduction variables. Which particular variable is updated depends on the value of *key* and is unknown at compile time. This prohibits efficient parallel execution and data partitioning on distributed-memory systems.

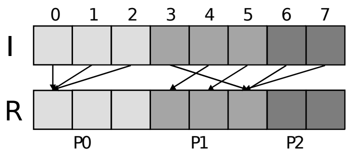
In general reductions can be distinguished by their data access pattern. Regular reductions exhibit an access pattern that is predetermined algorithmically. In this case, the reduction algorithm can be parallelized in a way that concurrent instances operate on disjoint memory locations thus avoiding race conditions. Additionally, the regular data access pattern allows an efficient distribution and computation on parallel architectures with disjoint memory address spaces (such as clusters, GPGPUs and accelerators) because no interprocess communication is required for its computation. Figure 5.2(a) shows a concurrent vector addition $R = R + I$, implemented as a regular reduction on input vector I and output vector R with $R[Ridx] = R[Ridx] + I[Iidx]$. The mapping function *map* shows a *static* relation between indexes generated by the control flow $Iidx$ (they index the input array) and output indexes $Ridx$ (indexing the output array) and is defined as $Ridx = map(Idx)$ with map being the identity function $map(x) = x$. In this example input and output data are distributed to processors $P0$, $P1$ and $P2$. Each processor operates on local data only.

On the other hand irregular reductions exhibit a data access pattern that is not determined by the control flow of the algorithm. Their access pattern depends on input data. In this case it is not possible to guarantee race-free concurrent execution algorithmically. Furthermore performance concerns arise.



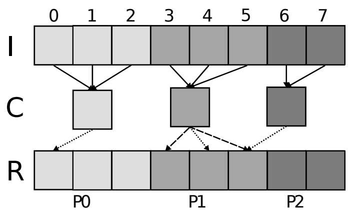
$R[Ridx] = R[Ridx] + I[Iidx],\ Ridx = map(Iidx)$

(a) Vector sum



$R[Ridx] = R[Ridx] + 1,\ Ridx = map(I[Iidx])$

(b) Histogram



$C[Cidx] = C[Cidx] + 1,\ Cidx = hash(map(I[Iidx]))$

(c) Cached histogram

Figure 5.2: Example of regular, irregular and cached reductions running in parallel on processors P1, P2 and P3

On SMP systems concurrent write accesses of shared data lead to mutual cache invalidations by cache coherence protocols resulting in performance degradation. Likewise locking techniques to implement atomic memory updates in order to ensure memory consistency potentially introduce high lock contention that leads to a degraded performance. On clusters, advanced techniques are needed to enable irregular reductions. For this purpose typically either data or computation are replicated to remote nodes (*ReplicateBufs* [3], *SelectPriv* [12] and *LocalWrite* [29]). Figure 5.2(b) shows a histogram computation on input vector I and an irregular reduction to output vector R as $R[Ridx] = R[Ridx] + 1$. The mapping function is input data dependent and defined as $Ridx = map(I[Iidx])$. In this case, node-local access cannot be guaranteed anymore.

5.1.2 Solution design

Our solution removes irregular updates from the critical path of the application through caching of output data. A thread-local cache serves as a temporal reduction target with all accesses being redirected to the cache. The cache takes care of fetching, storing and evicting cache entries. This approach guarantees conflict free updates on local nodes, helps to exploit both temporal and spatial locality and defers irregular updates to the point where evicted cache entries are committed. Cache evictions overlap ongoing computation thus effectively hide network bandwith and latency. Further this approach puts minimal requirements on front-end compilers thus making it a solution of choice for high programmability languages.

5.2 Language Support in OmpSs

We evaluate programmability and language requirements of *CachedPrivate* with OmpSs [26]. OmpSs is a high-level, task-based, parallel programming model supporting SMPs, heterogeneous systems (like GPGPU systems) and clusters. We chose OmpSs for several reasons. OmpSs is representative for a commonly used parallel programming paradigm based on tasking. Further, its modular plug-in based design makes new directives (such as work-sharing directives for implicit tasking) and conceptual work on user-defined reductions [30] easy to implement. Finally, its runtime features are beneficial for *CachedPrivate*, especially locality-aware task scheduling and dependency-aware task execution.

5.2.1 Tasking in OmpSs

task [clauses] function definition — function header

As discussed previously, the OmpSs *task* pragma marks a function for asynchronous execution and accepts *input*, *output*, *inout* and *concurrent* clauses to define data directionality. The input clause lists all task parameters with read-accesses, the output clause lists all parameters with write-access and inout clause lists those with read- and write-access within the task. The concurrent clause relaxes the true dependency between tasks and allows their parallel execution. In this case the developer is required to take care of correct synchronization.

The *target* clause specifies a target device such as an accelerator or SMP. This information allows the runtime to deploy task data to the target device automatically as required by the input and output clauses. In this case task input triggers a data copy from master (data owner) to device and an output a copy back from device to master. OmpSs further offers synchronization constructs through the *#pragma omp taskwait* and *#pragma omp taskwait on (var)* pragmas where *var* is a shared variable.

5.2.2 Manual reductions in OmpSs

Parallel reductions in OmpSs are currently programmed by explicitly creating tasks that perform them. Figure 5.3 shows a parallel histogram computation over the input *keys* and output *histo.* In this case histo represents a reduction variable. While the input data can be divided into chunks of size *block* for each task, the output data is passed to each task in full size due to irregular updates over the entire length. In order to avoid task serialization, the *concurrent* pragma is used to override task dependencies that would result from the *inout* declaration. This approach requires an atomic update on SMP systems (implemented here with a *#pragma atomic*). On clusters, OmpSs requires the creation of private copies of output array for each task. Once a task finishes, it copies back its private copy holding an intermediate result of the reduction operation to the originating node. This node then reduces all incoming privatized arrays and computes the final result. While privatization (ReplicateBufs) is a reduction scheme that can be language-supported and transparent to the developer (opposed to other approaches requiring substantially more programming efforts) it does not scale. Performance scalability is limited due to resulting high memory and network utilization and node local reduction of intermediate results. Figure 5.4 shows a parallel histogram implementation with privatization in OmpSs.

```
void Histo(TYPE* keys,TYPE* histo, int K, int H) {
  int start=0, block, block_mod;
  int NT = NUM_TASKS;
  TILESIZE (K, NT, block, block_mod);
  for(int i = 0; i < NT; i++)
  {
    if (i == NT-1) block += block_mod;
    #pragma omp target device(smp)
    #pragma omp task input (keys[start;block])  \
                      concurrent(histo[H])
                      firstprivate(start, block)
    {
      INT_TYPE key;
      for(int j = start; j < start + block; j++)
      {
        key = keys[j];
        #pragma atomic
        histo[key]++;
      }
    }
  start += block;
  }
}
```

Figure 5.3: Parallel histogram with locks

```
void Histo(TYPE* keys,TYPE* histo, int K, int H) {
   [...]
  TYPE** privHisto = Alloc(){...}
  for(int i = 0; i < NT; i++)
  {
    TYPE * _histo = privHisto[i];
    if (i == NT-1) block += block_mod;
     #pragma omp target device(smp) copy_deps
     #pragma omp task input (keys[start;block])  \
                        output(_histo[H])
                        firstprivate(start, block)
    {
      Init(_histo)
      INT_TYPE key;
      for(int j = start; j < start + block; j++)
      {
        key = keys[j];
        _histo[key]++;
      }
    }
    start += block;
  }
   #pragma omp taskwait
  Reduce(privHisto){...}
}
```

Figure 5.4: Parallel histogram with privatization

```
#pragma omp task [clauses] reduction (identifier:var)
function definition | function header
```

```
#pragma omp for reduction (identifier:var)
[clauses]
```

Figure 5.5: Reduction support in task and work-sharing language constructs in OmpSs

```
#pragma omp declare reduction (identifier :\
            type-list : combiner) [initializer]
```

Figure 5.6: User-defined reductions in OmpSs

5.2.3 Language support for reductions in OmpSs

For programmability we introduce the *reduction* clause for explicit tasks. It extends the OmpSs task construct and instructs the compiler to generate additional runtime calls that provide required information to the CachedPrivate runtime support. Particularly the front-end compiler invokes the following steps when encountering a reduction

- Generate API calls to CachedPrivate for initialization of supporting data structures
- Replace references to reduction variable with calls to CachedPrivate
- Introduce an implicit memory barrier on reduction variables to guarantee data availability when required (this is the case if a subsequent task has an input data dependency on reduction data or when a reduction is followed by a synchronization construct). Further this step is required to identify all manually created tasks that participate in the reduction.

Its definition is shown in Figure 5.5.

Further we propose the addition of the *pragma omp for* work-sharing construct to the OmpSs language front-end (in a work-sharing construct, tasks are created implicitly). The work-sharing construct accepts the reduction clause and is defined as shown in Figure 5.5. In order to make the reduction clause generic, we extend it by user-defined reductions (UDRs). This makes any user-defined function a reduction operator that is applicable on arbitrary reduction data types.

The expression of a UDR consists of a reduction declaration and usage. Figure 5.6 shows a reduction declaration where *identifier* names a reduction

```
void Add(int & a,int & b) {a+=b;}
void Init(int & a) {a=0;}
#pragma omp declare reduction(Add:TYPE:Add(omp_out,omp_in))\
 Init(omp_priv)

void Histo(TYPE* keys,TYPE* histo, int K, int H) {
  int start=0, block, block_mod;
  int NT = NUM_TASKS;
  TILESIZE (K, NT, block, block_mod);
  for(int i = 0; i < NT; i++) {
    if (i == NT-1) block += block_mod;
    #pragma omp target device (cluster) \
         copy_in(keys[start;block])
    #pragma omp task input (keys[start;block]) \
    firstprivate(start, block) \
    reduction(Add:histo[H])
    {
      TYPE key;
      for(int j = start; j < start + block; j++)
      {
        key = keys[j];
        Add(histo[key],1);
      }
    }
  }
  start += block;
}
```

Figure 5.7: Histogram with CachedPrivate and explicit tasking

operator, *type-list* lists supported data types, *combiner* extends the reduction operator by semantic information and *initializer* defines output data initialization. A reduction operator must be a binary, commutative and associative function implementing an update operation as expressed by the combiner. The combiner accepts *omp_out* and *omp_in* clauses that indicate the update semantics. The initializer follows the idea of the combiner and defines initialization semantics *omp_priv* and *omp_orig*. The reduction usage employs the reduction declaration in the reduction clause.

A parallel histogram with the new OmpSs reduction scheme with explicit tasking and with a work-sharing construct (implicit tasking) are shown in Figure 5.7 and Figure 5.8.

```
void Add(int & a,int & b) {a+=b;}
void Init(int & a) {a=0;}
#pragma omp declare reduction(Add:TYPE:Add(omp_out,omp_in))\
 Init(omp_priv)

void Histo(TYPE* keys,TYPE* histo, int K, int H) {
  #pragma omp for shared(keys[K]) \
  reduction(Add:histo[H])
  TYPE key;
  for(int i = 0; i < K; i++)
  {
    key = keys[i];
    Add(histo[key],1)l
  }
}
```

Figure 5.8: Histogram with CachedPrivate and implicit tasking

5.3 CachedPrivate

Caching offers the advantage of making an irregular reduction input-data independent, thus regular, by applying a fixed function map that redirects previously scattered memory accesses to a cache. This has several advantages. Firstly it allows the use of small cache sizes, making this approach memory-efficient. Further it allows us to exploit data characteristics such as input data sparsity. In case of sparse output, updates happen on a subset of reduction data only. This data is likely to be kept in cache throughout the computation. Consequently this maximizes data reuse and minimizes communication requirements. Also irregular accesses are deferred to a point later in time, typically when a cache entry is evicted and needs to be committed. Comparably to a CPU cache, where evicted cache entries are written back to the owning memory location, CachedPrivate communicates evicted cache entries to the owning process. On arrival the cache entry is committed to memory. It turns out that the time required for committing evicted cache lines can be effectively overlapped by computation. This can hide network bandwith and latency.

Once the computation finishes, all outstanding cached entries are flushed. Cache flushes cannot be overlapped and therefore are in the critical path of the reduction computation. Naturally for small cache sizes (256KB) this time is negligible. Figure 5.2(c) shows the caching approach where all update operations are redirected to thread local caches. The reduction operation is defined as $C[Cidx] = C[Cidx] + 1$ with C being the cached reduction data and index $Cidx$ computed as $Cidx = hash(map(I[Iidx]))$. The hash function defines a mapping of the direct-mapped cache. In our case we used the modulo operation. In the example, the cache is of size one and can hold one entry at a time. Consequently caching multiple addresses results in cache evictions. Cache evictions are marked with a dashed line whereas memory accesses during a flush are marked with a dotted line.

5.3.1 Runtime Support

The OmpSs runtime implements task management, the OmpSs execution and memory model and the CachedPrivate reduction API.

On application startup each participating thread initializes the runtime environment and creates a worker thread pool. One process becomes the master, all other processes become slaves. The master process begins main code execution and generates work descriptors for each encountered task declaration and work-sharing construct. Each time a work descriptor is created, the main thread adds a representing node to an internally managed dependency graph (a data-flow DAG) accordingly. If all dependency requirements of a task are satisfied, it is removed from the dependency graph and placed into the ready task queue. Worker threads poll the ready queue for ready tasks, process tasks asynchronously and update the dependency graph once a task execution is complete. Main code execution is suspended once a synchronization point is reached.

To execute tasks on remote cluster nodes, each remote process has a representative in the master node thread pool. While the representatives are not physical threads, they participate in polling activities. The polling of the ready queue is directed through the scheduling object, which applies a scheduling policy and returns a particular ready task based on a scheduling decision. OmpSs supports Round-Robin, priority based and locality-aware task scheduling.

```
void Histo(TYPE * keys, TYPE * histo, int K, int H) {
  [...]
  for(int i=0; i<NT; i++) {
    /*task generation mechanics*/
    addTarget(histo, localityMap, &Operator, &Initializer);
    {
      /*task body*/
      INT_TYPE key;
      for(int j = start; j < start + block; j++)
      {
        key = keys[j];
        INT_TYPE val;
        read(&histo[key],&val);
        Operator(val,1);
         write(val,&histo[key]);
      }
       flush();
    }
    start += block;
  }
}
```

Figure 5.9: Parallel histogram, as shown in Figure 5.7, after compiler translation to CachedPrivate

5.3.2 CachedPrivate implementation

We implemented CachedPrivate with a thread-local, direct-mapped, write-back software cache and the following API calls: *addTarget*, *read*, *write* and *flush*.

These API calls are added to the user code automatically by the front-end source-to-source compiler. Figure 5.9 shows user task code after compiler translation. *AddTarget* is called first and marks a task as a reduction task. Further it configures reduction target, locality map, reduction operator and initializer. The reduction target represents the original address pointer on the master node and is used, together with the locality map for address translation. The locality map is generated by the runtime and defines ownership of reduction data in case its blocks reside on remote nodes. Finally the reduction operator and initializer are function pointers to the respective function as provided by the user using the UDR language construct. Once a reduction task is scheduled for execution on a worker thread, this information is applied

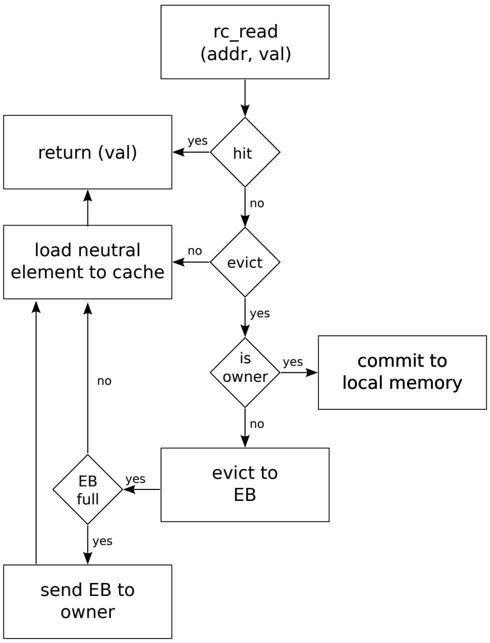


Figure 5.10: Reading from reduction cache

to the thread-local cache object. The cache now can initialize to the neutral element by applying the initializer function and is ready to start.

Read and *write* calls replace read and write accesses to the reduction variable in the original code. Figure 5.10 illustrates the control flow of the read API call. Every cached read operation results in a cache hit or cache miss. In case of a cache hit, the cached value is immediately returned and task code execution continues. In case of a cache miss the runtime verifies the state of the cache entry. If the state is marked as dirty, the cache entry is evicted. In case the entry is marked invalid, the runtime applies the initializer function, updates the cache entry with the neutral element and returns the new value.

Cache evictions are either committed to local memory (in case the address tag belongs to a block owned by the current node) or stored in an eviction buffer (*EB*). This buffer can hold up to a configurable number of evicted cache entries. Once it fills up it is sent to the owning process. The eviction buffer effectively implements eviction granularity, comparably to cache lines on CPU caches. Write returns a cached value. Since each cache is thread local, it can be safely assumed that a cache entry remains unchanged and therefore valid after a previous read. For this reason it is not required to verify cache entry state on a cache write operation.

The *flush* API call is invoked by the runtime when the execution of user task code has finished and cached data needs to be flushed. In that case all dirty marked cache lines are evicted as described above.

On remote nodes, incoming cache evictions are queued and wait for being processed. To speed up processing we introduced reduction threads. Reduction threads are similar to worker threads except that they are not included in task execution but are dedicated to processing reductions.

5.3.3 Defining reduction topology

In order to support heterogeneous environments with different properties (such as processing speed, network bandwith or latency), CachedPrivate supports different reduction topologies. A reduction topology consists of links and nodes that define a reduction path. Currently 1:N and M:N reduction topologies are supported. A particular reduction topology is defined by the ownership of the output reduction array and is independent from input data. In case of 1:N, the entire array resides on the master node and cache evictions are always sent to the master process. In case of M:N, the array was distributed beforehand (which often the case for parallel application) and its blocks reside on M nodes. In this case evicted cache lines are sent to the respective block owner. In an N:N configuration, data and communication are evenly distributed over a cluster which generally seems to be the topology of choice.

Figure 5.11 shows how the programmer can define ownership and the resulting reduction topology by a parallel initialization of *histo*. The resulting dependencies, defined by the *output(histo[start;block])* output dependency of the initialization tasks and the *input(histo[h])* input dependency of the reduction tasks are shown as a task dependency graph in Figure 5.12.

```
[...]
for(int i = 0; i < NT; i++){
  #pragma omp target device (smp) copy_deps
  #pragma omp task output(histo[start;block]) \
  firstprivate(start ,block)
  {
    for(int j = start; j < start + block; j++)
    histo[j] = 0;
  }
  start+=block;
}
#pragma omp taskwait noflush
```

Figure 5.11: Initialization tasks defining CachedPrivate reduction topology

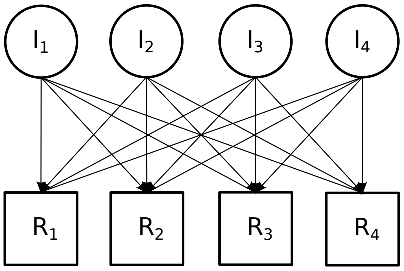


Figure 5.12: Task dependency graph with initialization and reduction tasks

5.4 Evaluation

5.4.1 Methodology

We evaluate CachedPrivate with the parallel histogram application, different input data characteristics and runtime configurations. We chose the histogram application for the following reasons.

- Its scalability relies mainly on memory access efficiency.
- Its input data directly relates to the memory access patterns. This allows us to configure different input data sets with different characteristics and to measure their impact on the presented approach.

- It represents a worst-case application scenario with very frequent irregular updates.

To analyze the impact of different input data characteristics, we apply the benchmark on different input data sets. We quantify an input data set by the resulting memory access pattern. Particularly we are interested in:

- Sparsity, defined as $1 - (R/So)$ with R being the number of updated histogram entries (reduction output) and So being the total number of histogram entries (size of output).
- Connectivity, describing the ratio of Si to R with Si being size of the input array (size of input).
- Node locality, representing the relation between local memory updates and memory updates on remote nodes.

Naturally sparsity and connectivity have a significant impact on caching efficiency and often require more advanced caching algorithms. In this work we do not apply advanced caching techniques even though we are aware of their significance. Instead, it turned out that a simple, direct-mapped cache is sufficient to investigate the runtime's ability to overlap cache line evictions with local computation, runtime overhead and the resulting scalability of this approach. By doing so we believe that this evaluation allows to estimate a worst-case scalability behavior of CachedPrivate for many other applications.

Further we investigate different runtime configurations by setting the following properties.

- 1:N and N:N reduction topologies
- Number of worker threads
- Number of reductions threads

5.4.2 Environment

All benchmark runs were performed on 32 cluster nodes running Red Hat 4.4.4. Cluster nodes are interconnected by an Infiniband network and 14 switches. Every node has two Intel Xeon E5649 6-Core CPUs running at 2.53 GHz with 24 GB of RAM. Applications were compiled with the OmpSs Mercurium C++ compiler v1.3.5.8 and GCC v4.4.4 back-end compiler with -O3. The runtime is based on OmpSs Nanos++ v0.7a. We used the Berkley GASNet distribution v1.18.2 for single-sided MPI interprocess communication configured with OpenIB (ibv) driver support, pthreads support and segment-fast configuration.

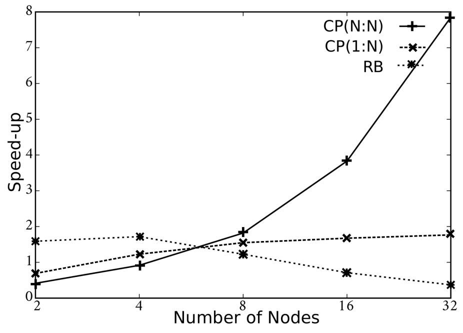
5.5 Results

The following results are based on input data with low sparsity as well as connectivity. The runtime was configured with a cache size of 2^{16} elements (256KB). Cache evictions are committed in a single-node (1:N) and multiple-node (N:N) reduction topology. For scalability benchmarking we use a 2GB large, dense, integer array as input, initialized with uniformly distributed random numbers over a large range. The output histogram array is 512MB in size and holds one entry for each possible value. This input set is loosely connected, with update operations scattered over the entire histogram array with 4 updates per memory location on average. Loose connectivity results in low cache efficiency because each cached address is accessed only 4 times on average during the entire computation. This makes the probability of reusing a cached line for most caching algorithms negligible.

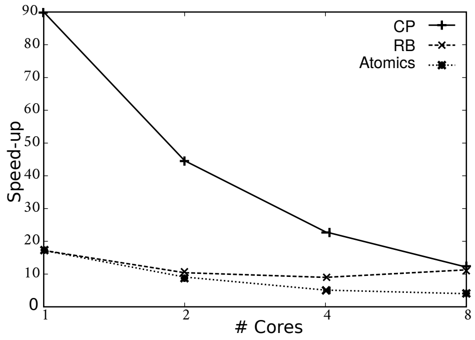
5.5.1 Node scalability

Figure 5.13(a) shows histogram scalability with one worker thread per node for ReplicateBufs (privatization) and for CachedPrivate with both topologies. The baseline represents serial code performance. Data distribution time, that is the copy-in time is not included in the results.

The 1:N reduction topology quickly hits scalability limitations. This is due to the limited single-node performance that cannot process evicted cache lines from N-1 processes sufficiently fast. The N:N reduction topology shows better scalability. In this case cache evictions are uniformly distributed over all participating nodes and can benefit from a scaling number of nodes. As expected ReplicateBufs does not scale.



(a) Node scalability



(b) SMP scalability

Figure 5.13: Node- and SMP scalability on dense, loosely connected input data (2GB input, 512MB output, 256KB cache and eviction buffer sizes) with CachedPrivate (CP) and ReplicateBufs (RB) with one worker thread per node and atomics

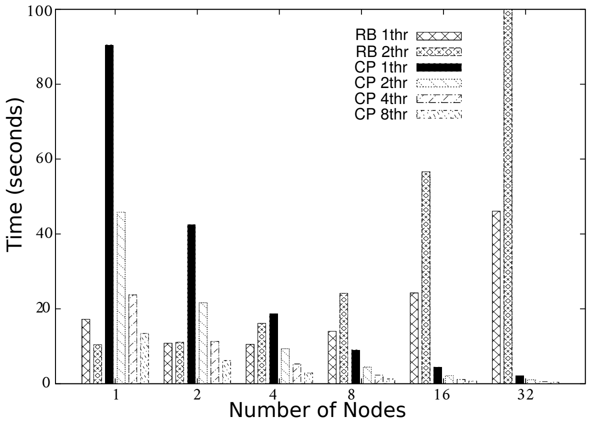


Figure 5.14: Worker thread scalability of CachedPrivate (CP) and ReplicateBufs (RB)

5.5.2 SMP scalability

In order to validate that the achieved cluster performance scales beyond the performance boundaries of a SMP system, we compare CachedPrivate with ReplicateBufs and atomic updates on a single cluster node. Figure 5.13(b) shows that single-thread performance of CachedPrivate is dominated by the introduced overhead. By adding more CPU cores to the computation, performance results become comparable to those obtained with ReplicateBufs. Atomic operations suffer from lock contention and achieve a speed-up of 4x.

5.5.3 Reduction thread scalability

Figure 5.14 shows how CachedPrivate and ReplicateBufs adapt to a scaling number of worker threads. While ReplicateBufs quickly degrades in performance due to heavy network utilization and the reduction of intermediate results, CachedPrivate reaches a speed-up of 43x on a fully utilized cluster (32 nodes with 8 threads per node).

5.5.4 Impact of other parameters

We conducted further benchmarks with higher degrees of sparsity and connectivity. As expected, the application performance did not change noticeably. This is due to the currently employed direct-mapped cache that does not perform well for random accesses over large address ranges that do not fit into cache.

Further we tested input data sets with different node localities. Results confirm that the performance with different node localities is comparable to results obtained in the 1:N and N:N reduction topology evaluation. Particularly 1:N represents a worst-case scenario with very low node locality on N-1 nodes and N:N representing an average-case scenario with each node owning one N^{th} part of the output array. Also we observed that the presented caching approach adapts well to changing memory access patterns between iterations. This is given by the very small ratio of cache size to output vector size that allows the cache to adapt to any access pattern rapidly.

5.6 Conclusion

In this chapter we presented programmable and scalable support for irregular reductions. We demonstrate its programmability in the OmpSs parallel programming model for both explicit task and work-sharing costructs. By using a pragma annotation, the programmer can easily express a reduction and take advantage of available parallel hardware. Runtime scalability is achieved through software caches that allow overlapping communication and computation and maximize temporal and spatial data reuse. The proposed approach requires a minimal API and is applicable on other programming models as well.

Experimental evaluation shows that with a small coding effort, a highly irregular histogram computation can scale up to 32 nodes on a cluster. Best results are achieved when reduction output data is distributed among cluster nodes in an N:N reduction topology.

Despite the fact that the organization of our software cache is simple, results are promising. We plan to continue research and development of *CachedPrivate* and further focus on caching efficiency (using different organizations and replacement policies), reduction efficiency (by supporting other reduction topologies such as reduction trees) as well as hardware support.

Chapter 6

Supporting Irregular Array-type Reductions on Shared Memory Systems with PIBOR

While software caching can reduce communication traffic for distributed irregular array-type reductions on cluster level, techniques are needed to achieve scalability on node level as well. In this chapter we present privatization with in-lined, block-ordered reductions (*PIBOR*), a new approach that trades processor cycles to increase locality and bandwidth efficiency for such algorithms. A reference implementation in OmpSs shows promising results on current multi-core systems.

6.1 Introduction

Irregular array-type reductions are cache inefficient due to poor data locality. Furthermore, to avoid data races where multiple threads perform an update of a single memory location at the same time, accesses either need to be synchronized (via memory barriers or thread synchronization such as atomics), ordered or redirected.

Access redirection to a thread-private copy of the reduction target eliminates the need for access synchronization. Unfortunately, while it works well for scalar types, it becomes expensive for arrays and useless for large data sets. Figure 6.1 shows the performance impact of array privatization and atomics on scalability in the RandomAccess [7] benchmark running with 16 threads and different problem sizes. For reference, the figure also shows an implementation with data races caused by unprotected concurrent accesses (*Race*). Source codes for atomics and privatization are shown in Figure 6.2.

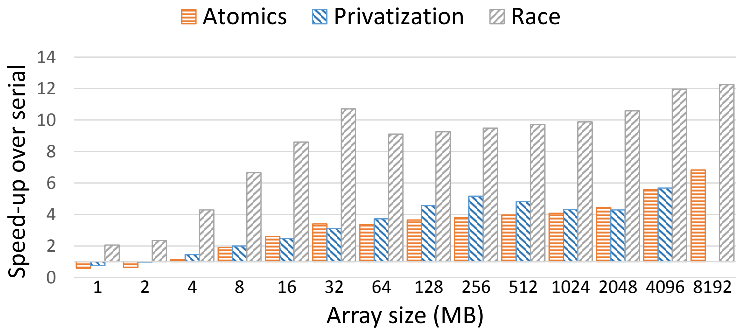


Figure 6.1: Speed-up of the RandomAccess benchmark on the Intel Xeon E5 processor with 16 threads implemented with different parallelization techniques.

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 redirecting accesses to thread-private, linear buffers that correspond to memory regions of the reduction array and flushing the buffers when they are full, is a simple yet efficient technique to meet the above requirements. The contributions of this work

```
int Table[S];
int _Table[max_threads][S];
init(_Table);
for(j=0; j<num_tasks; j++){
 uint64_t seed = ran[j];
 #pragma omp task \
               shared(_Table)
 {
  for(i=0;i<block; ++i ){
  uint64_t pos=getpos(&seed);
  k= omp_get_thread_num();
  _Table[k][pos] ^= seed;
}}}
#pragma omp taskwait
reduceTable(_Table, Table);
```

(a) Privatization

```
...
int Table[S];
init(Table);
for(j=0; j<num_tasks; j++){
 uint64_t seed = ran[j];
 #pragma omp task \
               shared(Table)
 {
  for(i=0;i<block;++i){
  uint64_t pos=getpos(&seed);
   #pragma omp atomic
  Table[pos] ^= seed;
}}}
#pragma omp taskwait
...
```

(b) Atomics

Figure 6.2: Concurrent RandomAccess kernel benchmark showing two common parallelization schemes.

are the introduction of PIBOR, its evaluation and its proposal as runtime support to accelerate array reductions on fast processors. This work is also part of our initiative to propose and evaluate advanced techniques to support array reduction in the upcoming OpenMP [1] standard.

6.2 Implementation

To support irregular array-type reductions, we developed a new technique called PIBOR. In this approach memory accesses to the original reduction array are redirected to a thread-private buffer and the update operation is replaced by an assignment of the right-hand side of the reduction expression. Unlike 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 full, the owning thread reduces the buffer to global memory by applying the reduction operation.

Typically writing out data to global memory in parallel requires to set a global lock over the entire data structure which serializes execution. We prevent this by dividing the reduction array into disjoint regions and by allocating multiple buffers on each thread where each buffer corresponds to a memory region of the reduction array. In this way accesses to the original reduction array that would fall within a certain region are stored in the corresponding buffer on the executing thread. In case a buffer is filled up, 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. In order to avoid busy-waits in case a region is locked, the requesting thread skips to the next region that has reached a sufficient fill. A schematic overview of an application that runs N tasks on N threads and performs a reduction over an array divided into M regions is shown in Figure 6.3.

Since memory accesses fall into different regions, each memory access needs to be inspected in order to determine its corresponding region and buffer. We do so by applying the hash function $pos = f(addr) = (addr - array\_start) >> log_2(region\_size)$ on the address of the accessed element. The hash function takes an address as argument, normalizes it to zero and right-shifts the value to return the most-significant bits that determine the region number (which is equivalent to a division of array size by region size).

Arguably, this approach introduces an instruction overhead relative to original codes that often are as simple as a load, followed by an increment and store operation. However, it turns out that using processor cycles in order to replace scatter-updates over an entire array by linear writes into an array of buffers followed by a scatter over a region (of which size can be optimized to reflect architectural and system properties), leads to higher execution performance due to higher locality. On fast processors with slow memory (sometimes referred to as memory gap), the instructions overhead becomes less significant. A control flow diagram of entire process is shown in Figure 6.4.

6.2.1 Language support

To support PIBOR, the front-end compiler is required to replace all occurrences of the reduction array and its operator by a reference and assignment to a buffer. Figure 6.5 shows a simple case, where the compiler introduces a temporal variable and inserts calls to the runtime to acquire storage (*_mem_request(reduction identifier, operator)*) and buffer entry

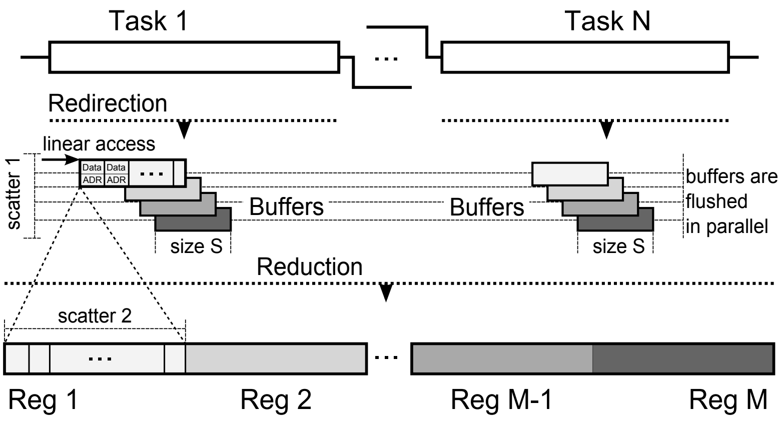


Figure 6.3: Schematic view of PIBOR showing access redirection into buffers (*scatter 1*) and regions of configurable size (*scatter 2*) avoiding scatter-updates over large memory.

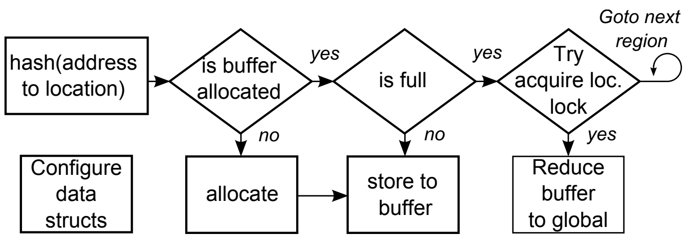


Figure 6.4: Execution diagram of PIBOR.

(*_pos_request(address, temporal variable, storage)*). For performance reasons, the request for buffer position is implemented as an in-lined header function.

The presented compiler support however does not allow name aliasing and is restricted to the lexical scope. Name aliases and calls to external functions would result in memory accesses violations since they expect the original memory layout. This is not a restriction induced by PIBOR, but common to all approaches that implement a different memory layout rather than creating full thread-private data copies. Consequently, we propose this approach either as an optimization for parallelizing compilers or as a runtime library used by

```
#pragma omp task reduction \
                (+:array[0;S])
{
  ...
  array[pos_N]+=(T)expression_N;
  ...
  ...
}
```

(a) User code

```
{
  pibor_t * P=_mem_request(array, op);
  ...
  T * _t;
  _pos_request(&array[pos_N], _t, P);
  *_t = expression_N;
  ...
}
```

(b) Generated code

Figure 6.5: Language support for array-type reductions in OmpSs.

```
#pragma omp task reduction (+:array[0;S])
{
  //use of alias
  int * alias = array;
  alias[pos_N] += expression_N;
  //call to external function
  f(array);
}
```

Figure 6.6: Unsupported use-cases with reductions performed over aliased variable names and in external functions.

programmers. Figure 6.6 shows the unsupported use-cases of aliasing and external function calls.

6.2.2 Optimization to support unbalanced access patterns

Different applications exhibit different memory access patterns within their reduction kernels. One particular pattern that represents a worst-case scenario for all approaches shown in Chapter 3.5 are hot-spots. Hot-spots are access imbalances where many updates are performed over a small number of memory locations. In PIBOR this would lead to a high lock contention over the involved memory regions, practically serializing execution.

In PIBOR sparse problems with execution hot-spots are supported by increasing the number of regions. Since each new region requires an additional buffer, we implemented on-demand allocation of buffers for threads and regions. In this case, even though many regions are defined, buffers are not allocated for unreferenced memory regions outside execution hot-spots. Similarly, buffers are not allocated for threads that do not participate in the reduction. On-demand allocation allows to set a fine-granular region resolution while maintaining the same memory footprint for unbalanced memory access patterns.

6.3 Evaluation

We evaluate PIBOR with RandomAccess and a prototype implementation in OmpSs. RandomAccess is a kernel benchmark that performs an irregular array-type reduction. Due to its inherently low data locality, it is the standard benchmark for memory subsystems, making it especially interesting to showcase performance gains through locality improvements. Further, its simplicity allows for a convenient simulation of different problem sizes and access patterns that mimic properties of other applications.

6.3.1 Methodology

For the purpose of evaluation we conducted experiments to explore parameter settings for PIBOR, its relative performance compared to implementations with atomics and privatization as well as its behavior for worst-case access patterns with access hot-spots. In PIBOR, three parameters can be configured: total size of allocated memory, size of buffer and size of memory region. The size of a memory region determines the number of regions and consequently how many buffers per threads will be allocated. The total size of allocated memory is computed as $mem\_total = (S * M) * num\_threads$, where S is the buffer size and M is the number of regions defined as $array\_size/region\_size$. In our implementation, always two out of three parameters can be set, while the third is computed automatically. This allows configurations for strategies such as limiting maximal memory use or to configure region and buffer sizes to match hardware properties. We conducted experiments for serial and parallel executions.

6.3.2 System configuration

Our benchmark results were obtained from the IBM POWER8 system and the MareNostrum supercomputer, both located at the Barcelona Supercomputing

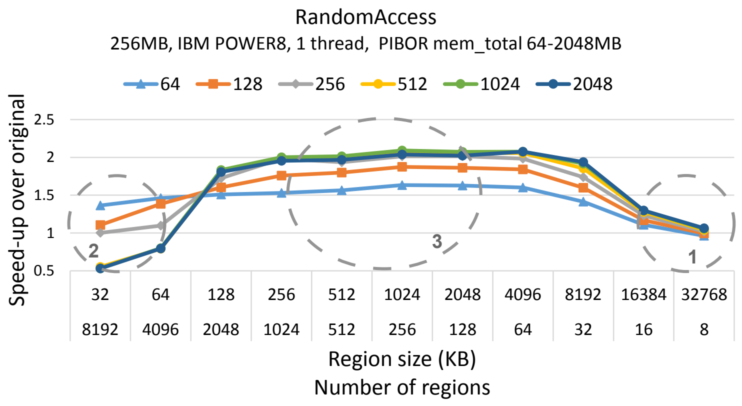


Figure 6.7: PIBOR parameter exploration on a single thread of the IBM POWER8 system shows different speed-ups over original code.

Center. The POWER8 is a quad-socket configuration with each socket containing 6 cores with 8MB of L3 cache and 8 threads (SMT) each. The chip is clocked at 2.1GHz and includes a 8MB L3 cache. The system has 1TB of memory with a bandwidth of 230 GB/s. The MareNostrum system node contains two 8-core Intel Xeon E5-2670 CPUs running at 2.6 GHz with 20MB L3 cache and 32GB of main memory with a bandwidth of 51.2 GB/s. The Intel Xeon Phi (7120P) features 61 cores at 1.23GHz and 30.5MB of L2 cache, with 4 threads per core and 16GB of memory with a bandwidth of 352 GB/s. Applications were compiled with the OmpSs Mercurium compiler v1.99 and GCC v4.8.2 or Intel C Compiler 15.0.2 as native compilers with -O3. The runtime is based on the Nanos++ RTL v0.9a.

6.3.3 Serial performance

To explore configuration parameters for PIBOR, we conducted experiments with variation of region sizes and total allocation sizes for single-threaded executions.

Figure 6.7 shows the speed-up over the original code on the IBM POWER8 system for six different allocation sizes and different region granularities. It shows that for small region counts (group *1* in Figure 6.7) execution

System	Size	Speed-up	#Regs	RegSize(M)	BufSize(S)	TotalMem
POWER8	256MB	2.1	256	1024KB	4096KB	1024MB
Xeon E5	256MB	1.1	32	8192KB	32449KB	1024MB
Xeon Phi	256MB	1.26	256	256KB	2048KB	1024MB

Table 6.1: PIBOR configurations that achieve the highest speed-ups for single-threaded executions on different architectures

performance drops since in this case each buffer corresponds to a large memory region (32768KB) and its reduction results in scattered updates over large data (*scatter2* in Figure 6.3). The memory access pattern in this case converges to that of the original code. In the opposite case, defining small regions (32KB, group *2* in Figure 6.7) results in introducing many regions and consequently many buffers. Also in order to accommodate many buffers into a predefined allocation space, each buffer becomes smaller. The additional handling of small buffers due to frequent buffer reductions as well as locality effects when accessing buffers located at distant memory location (*scatter1* in Figure 6.3) results in a performance drop.

It turns out that performance gains are highest (group *3* in Figure 6.7) when region size and total allocation size are chosen such that the resulting buffer size is a multiple of the regions size. This configuration achieves the highest local data reuse when reducing a buffer to memory. Table 6.1 shows configurations used to achieve the highest serial speed-ups on all three architectures for a given problem size.

Figure 6.8 shows speed-ups and memory allocation (*mem_total*) for different problem sizes on all three processors. On the Xeon E5 PIBOR does not achieve significant speed-ups. In single thread execution, the E5 has the highest memory bandwidth of all presented architectures. This reduces the ratio of processor speed to memory bandwidth which consequently reduces the performance benefits of PIBOR.

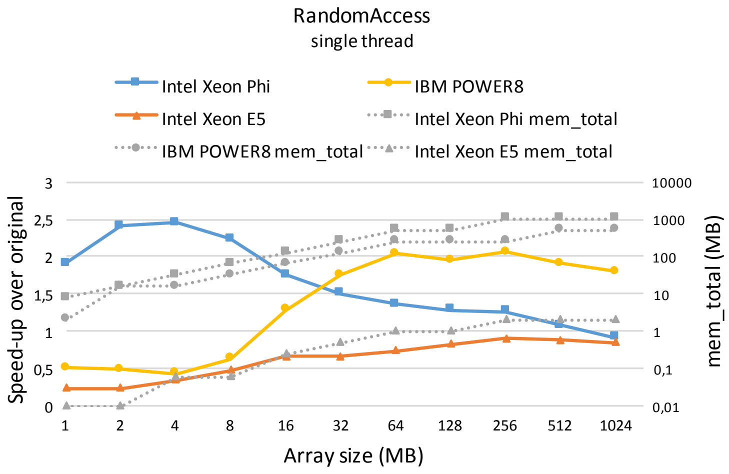


Figure 6.8: Execution speed-ups with PIBOR for single-threaded execution on three different system architectures.

6.3.4 Scalability

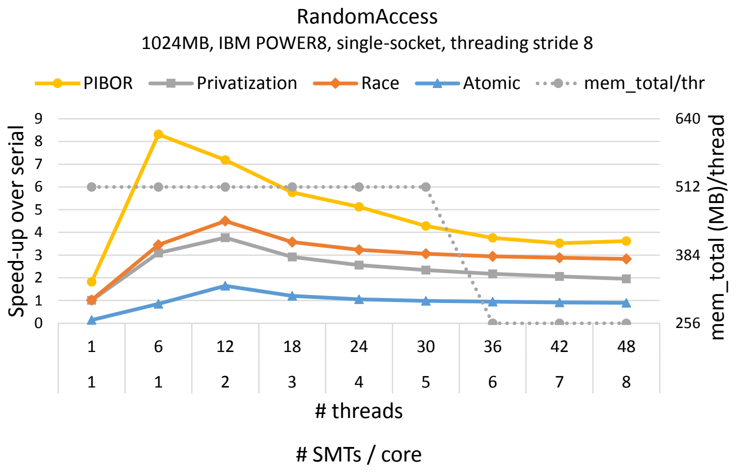
Parallel execution with PIBOR requires to tune region sizes to generate enough regions to avoid lock contention as well as to accommodate data of multiple threads within a shared cache.

Figure 6.7 shows performance scalability of PIBOR, atomics and privatization on the IBM POWER8 system for single-, dual-, and quad-socket executions and with thread striding of 8. This configuration binds threads to cores first, skipping SMT threads (8 per core), which results in performance peaks for multiples of 6 due to exclusive use of processor cache and bandwidth by a single thread per core. On all socket configurations, PIBOR achieves the highest speed-up due to improved data locality. The figure also shows the declining PIBOR total allocated memory per thread. Interestingly, on the IBM POWER8 quad-socket configuration, atomics outperform all other implementations for larger thread counts.

Figures 6.8 and 6.9 show scalability on the Intel Xeon E5 and Intel Xeon Phi coprocessor. PIBOR does not perform well on the Xeon Phi coprocessor

System	Size	Speed-up	#threads	#Regs	RegSize(M)	BufSize(S)	MemTotal
POWER8	1024MB	15.7	24	1024	1024KB	512KB	12288MB
Xeon E5	1024MB	8.1	16	128	8129KBs	2KB	4MB
Xeon Phi	256MB	33.3	56	64	4096KB	146KB	128MB

Table 6.2: PIBOR configurations that achieve the highest speed-ups for multi-threaded executions on different architectures

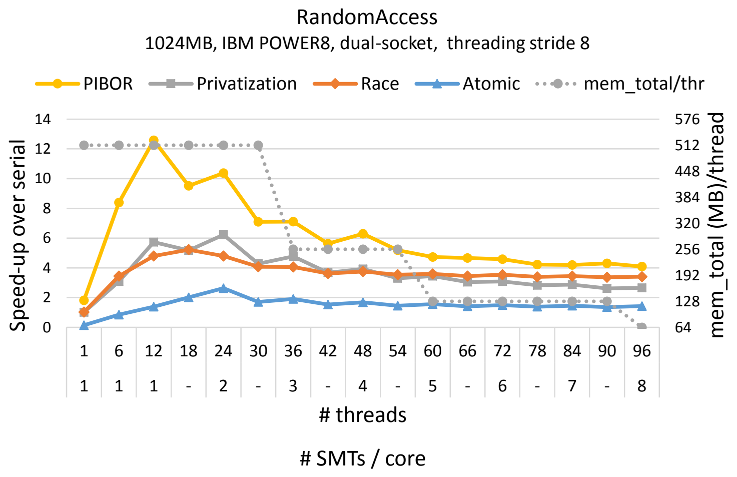


(a) Single socket

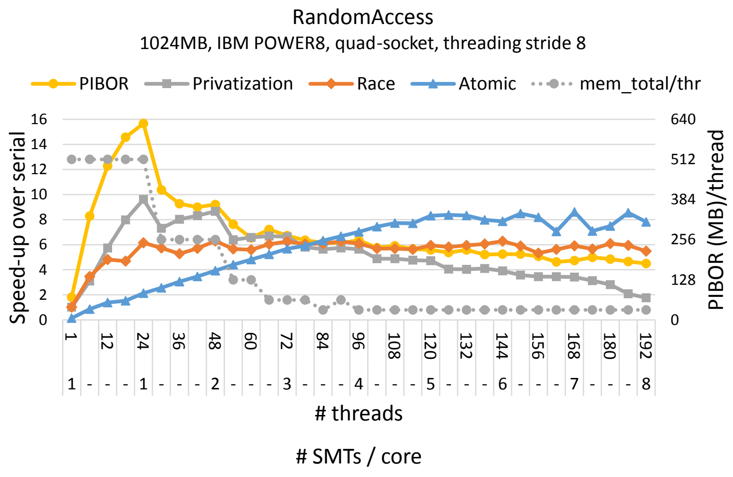
for executions with many threads. In this case processor speed relative to the aggregate memory bandwidth of all memory controllers is low and the introduced instruction overhead becomes more dominant than bandwidth limitations. Further the use of atomics that enforce in-order execution comes at no cost, since the Xeon Phi does not support out-of-order execution. The impact of atomics is visible on the POWER8 (especially for the single- and dual-socket executions) and Xeon E5 processors. Table 6.2 shows PIBOR configurations that achieved the highest parallel speed-up.

6.3.5 Memory access patterns

We evaluated the impact of memory access hot-spots by configuring the random number generator of the benchmark to generate indexes within a



(b) Dual socket



(c) Quad socket

Figure 6.7: Scalability of RandomAccess with PIBOR, atomic and privatization on different socket configurations on the IBM POWER8 system.

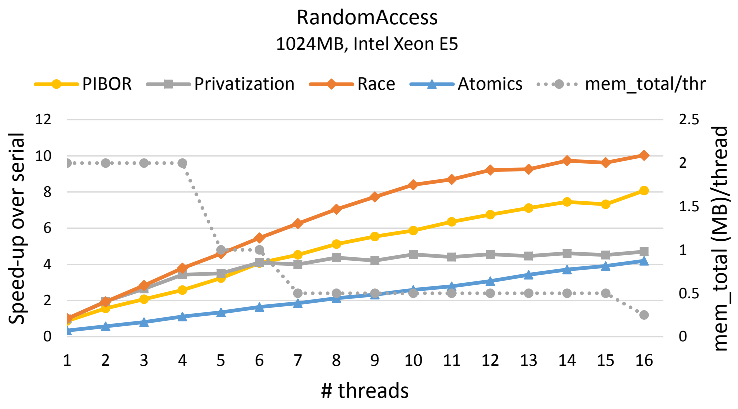


Figure 6.8: RandomAccess scalability on the Intel Xeon E5 processor.

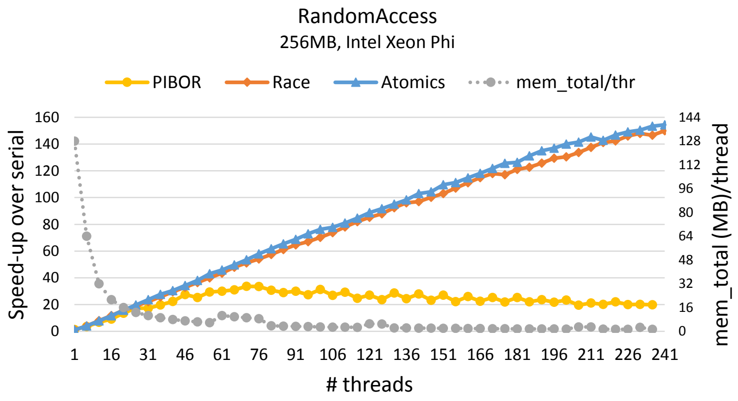


Figure 6.9: RandomAccess scalability on Intel Xeon Phi.

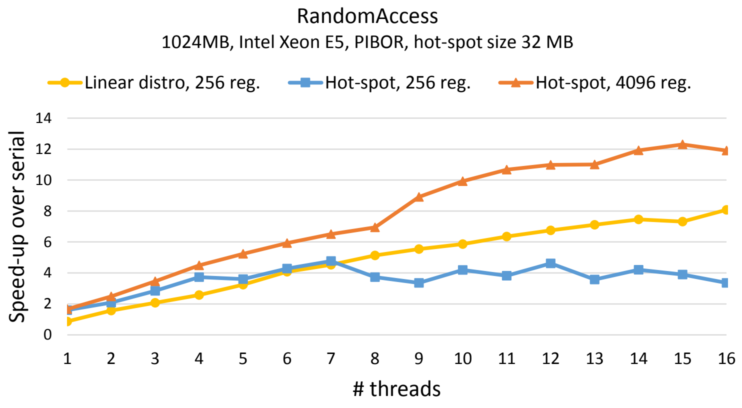


Figure 6.10: Access hot-spots are supported by reducing region size which results in an increased number of regions and less contention on region locks.

small range of 32MB. This version of the benchmark application is more cache efficient and results in a slight performance increase over the original code. Figure 6.10 shows scalability degradation due to increased lock contention for the hot-spot configuration where many threads update data within a small number of regions. By decreasing region size and increasing the number of regions from 256 to 4096, this effect can be mitigated.

6.4 Conclusion

The presented approach improves execution performance of irregular array-type reductions by increasing data locality through redirecting previously random accesses into a linear buffer. Full buffers are reduced to corresponding global memory regions which leads to higher memory reuse due to (region) local memory updates. PIBOR supports both, for- and while-loop reductions making this approach generally applicable and requires minimal extensions to parallelizing compilers. Performance results on Intel Xeon E5 and IBM POWER8 show that PIBOR outperforms implementations with atomics and privatization for single and multi-threaded executions and that a system's memory gap can be reduced by using spare cycles on fast processors to optimize for higher memory locality. Intel's Xeon Phi features a relatively slow processor with fast memory and in-order execution which allows for efficient use of atomics. Future work is directed towards automatic parameter tuning based on the underlying hardware architecture.

Chapter 7

Towards Unified Support for Array-type Reductions

Supporting irregular array-type reductions on modern systems is non-trivial as their irregular memory access pattern prohibits an efficient use of the memory subsystem and costly techniques are needed to eliminate data races. Taking a closer look at algorithms, memory access patterns and support techniques reveals that a one-size-fits-all solution does not exist and a solution is needed that gives the programmer the possibility of choice. Further, recent techniques require knowledge of access patterns. Inspecting such dynamic properties requires the addition of an inspector-executor to the programming model.

In this chapter we propose the OmpSs Reduction Model, a solution framework that generalizes the concepts of access redirection and iteration ordering. The goal is to enable support for a variety of techniques and allow programmers to switch between them without complex source code modifications. Further, we show an implementation of an inspector-executor and discuss which language extensions are needed. Our reference implementations use *PIBOR*, *PIBOR with Selective Privatization* and *Commutative Reductions* as representative techniques for both concepts.

7.1 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 such as irregular array-type reductions. 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. Further, techniques are needed to ensure correctness by avoiding data races that occur because of concurrent accesses of shared memory.

These properties led to the development of different support techniques that follow the strategies of either access redirection or iteration ordering. Access redirection is a strategy where accesses are redirected to a scratch memory while leaving the iteration space untouched. The scratch memory is either a thread-private copy of the original data (replication) or any other private storage that serves the same purpose but implements an *alternative memory layout*, *AML*. Ordering is a strategy that avoids redirection and reorders iterations to obtain a desired memory access pattern instead, thus creating an *alternative iteration space*, *AIS*.

Interestingly, techniques can be improved if information about the memory access pattern of the reduction kernel is provided to them. This creates the demand for programming models that are capable of inspecting dynamic properties and switching to optimized execution at a certain point in time. This execution model is called inspector-executor and works for cases where the executor can be run multiple times such that the benefit hides the cost of inspection. Adding inspector-executors to parallel programming models is not trivial since syntactical means are needed to express *what* variable to target, *when* to inspect and *how* to optimize.

Figure 7.1 shows a schematic representation of a reduction kernel where a global loop drives the progress of a simulation and performs an array reduction in each step. This iterative nature of kernel execution is a prerequisite for a meaningful use of inspector-executors. Figure 1.2, mentioned in the first chapter and Figure 7.2 show examples of such kernels. They originate from the LULESH [2] and SPECFEM3D [5] applications.

```
int i, res[S];
...
while(simulation_runs()){
  ...
  for(i=0; i<N; i++){
    res[f(i)]++;
  }
}
```

Figure 7.1: Structure of a typical scientific application with an array reduction in a global simulation loop

```
...
int i,j,k,iglob, elem;
  for (elem=0; elem<actual_size; elem++) {
    for (k=0;k<NGLLZ;k++) {
      for (j=0;j<NGLLY;j++) {
        for (i=0;i<NGLLX;i++) {
          iglob = ibool[elem][k][j][i];
          accel[iglob][X] += sum_terms[elem][k][j][i][X];
          accel[iglob][Y] += sum_terms[elem][k][j][i][Y];
          accel[iglob][Z] += sum_terms[elem][k][j][i][Z];
        }
      }
    }
  }
}
```

Figure 7.2: Representative kernel from SPECFEM3D application implementing an irregular array-type reduction over an array that is called iteratively within a global simulation loop

In this chapter we present the OmpSs Reductions Model (OmpSs-RM). It aims to enable generic support of reductions implementing AMLs and AIS for improved scalability of irregular array-type reductions. OmpSs-RM consists of a common interface, an inspector-executor and language support. The language specification allows programmers to select a particular technique without the need of code rewriting. The common runtime interface allows vendors to implement any privatization technique. The OmpSs inspector-executor tracks memory accesses and exposes access statistics to techniques that require them. To show-case this solution, we provide *PIBOR*, *PIBOR*

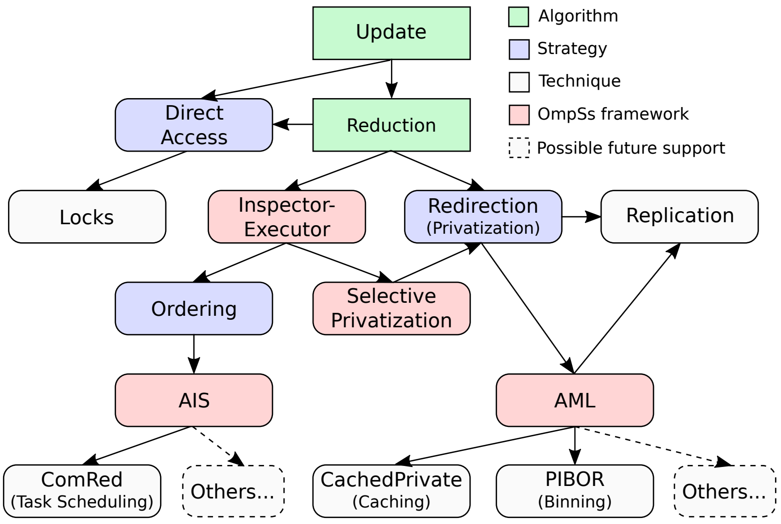


Figure 7.3: Landscape of algorithms, parallelization strategies and techniques

with Selective Privatization and *Commutative Reductions* (abbreviated as *ComRed*) as reference implementations of reductions with AMLs and AIS.

Results show that PIBOR with selective privatization and commuative reductions based on statistical data provided by the inspector achieve a substantial performance increase compared to other techniques.

Figure 7.3 gives an overview on how algorithms, parallelization strategies and techniques are related to each other. Further it shows which techniques require an inspector-executor as well as their support in the OmpSs reduction model.

7.2 Generalization through Reductions with AMLs

The most common technique to support reductions is data replication. This technique falls into the category of access redirection where memory accesses to the original reduction variable are replaced by accesses to thread-private copies of the original data in each task.

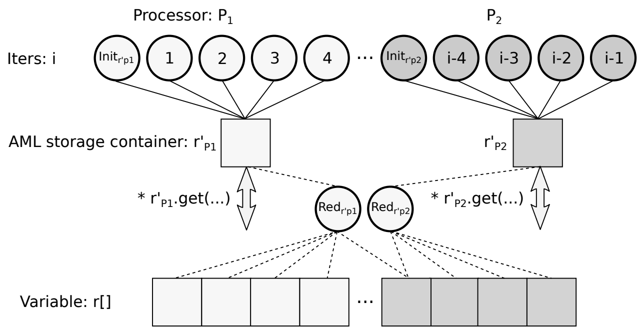


Figure 7.4: Redirecting accesses into thread-private data containers gives implementors the freedom to implement any arbitrary data placement strategy

Other techniques exist that follow the same idea of access redirection but implement an optimization strategy to improve data access locality of irregular array reductions. Representative approaches for this group are caching (*CachedPrivate*) and binning (*PIBOR*) as described in previous chapters. Software caches are useful in distributed memory scenarios where the cost of communication between nodes is higher than the processing overhead on the local node and where the problem is at least partially cache-friendly. Binning redirects accesses into bins that correspond to memory regions of original reduction array. Once a bin is filled, it is reduced to the corresponding memory location. The proximity of data within a bin results in an improved locality during the reduction phase. These techniques have three characteristics in common:

- Implementation of an alternative memory layout (AML) to accommodate thread-private data structures of a particular approach and to minimize the memory footprint

- Implementation of a custom initializer and reducer. These methods are invoked by the runtime.

- Implementation of a *get-method* that returns a pointer to a private storage that accommodates at least one reduction array element

In terms of AMLs, array replication is a case where the initializer and reducer methods operate on array copies and the get-method is obsolete as the [ ] operator can be used instead.

Figure 7.4 shows the generalization of access redirection where accesses to the original reduction variable are redirected into thread-private storage containers ${r'}_{P1}$ and ${r'}_{P2}$. Their allocation and initialization takes place on demand but before the execution of the reduction task code. In the Figure, initialization is marked as $Init_{r'p1}$ and $Init_{r'p2}$ and can be performed in parallel. Once the storage container is ready, execution of the reduction loop can start. For each memory access, the AML storage container provides a get-method that returns a pointer to a memory location. Which particular memory location is returned is implementation-dependent. The replacement of the original reduction variable by the get-method is a responsibility of the compiler at compile time. Once the execution of all reduction task is completed, outstanding data from storage containers is flushed into memory by applying the reducer functions $Red_{r'P1}$ and $Red_{r'P2}$. Which particular locations are accesses and in which order is implementation-dependent. For this example, the original access pattern is shown in Figure 1.3(b)

7.2.1 AMLs in OmpSs

Interface definitions as used in our reference implementation are as follows:

- void initializer(AML_T * priv, void * global)
- void reducer (AML_T * priv, void * global)
- void * get (void * address, void * global, analytics_info_t * a)

This interface allows runtimes to allocate any private storage of the size of an AML container (*sizeof(AML_T)*) and to invoke the initializer and reducer for each thread-private copy. The initializer method accepts a reference to an allocated thread-private storage as well as a reference to a global object. The global object is typically either a pointer to global reduction variable or any other structure containing additional information needed by the initializer such as size information. Similarly, the reducer function receives a pointer to a an AML as well as pointer to a global data structure containing further information if required by the reducer. The get-method returns a pointer to a private storage location to store a single target array element. It accepts a global object and an analytics object. The analytics object is computed by the inspector-executor on a phase switch. An example implementation is shown in the next section.

7.2.2 Handling AMLs by the Compiler

Our sample AML implements binning with selective privation through the aforementioned interface methods and its source code is located in a header file. During compilation, this header is inserted into the user application and accesses to the original reduction variable are replaced by calls to the get-method. Encapsulating AML techniques into header files is a design that offers support for any other user or vendor provided AML implementations without the need to recompile the programming model runtime. Further, due to the frequent accesses to the get-method, its in-lining into intermediate code reduces the overhead of stack operations.

Since reducers and initializers often require additional information about the reduction object such as array size or size of data type, additional parameters are passed. For this purpose, both functions accept a *reduction_info_t* object holding this information. In addition, the get-method takes a parameter of type *analytics_t* containing access analytics obtained during the inspection phase. Figure 7.5 shows the particular interfaces used to support AMLs in OmpSs.

Figure 7.6 shows the compilation process with the OmpSs front-end compiler (mcxx) and code organization as used in our implementation. Each header file contains implementations of the initializer, reducer and get-method for the particular AML technique. The generic runtime header (*runtime.h*) includes the definitions of *red_info_t*, *analytics_t* and the log-method used during inspection.

7.2.3 Handling AMLs by the Runtime

The runtime support of OmpSs-RM for AMLs builds on top of the existing reduction support where the runtime is capable of registering a new reduction

```
//AML interface
inline void * AML_GET (void*adr, AML_t*aml, analytics_t*mal);
void AML_init (AML_t * priv, reduction_info_t * info);
void AML_red (AML_t * in, reduction_info_t * info);
```

Figure 7.5: Interface to an AML object that implements binning with selective privatization

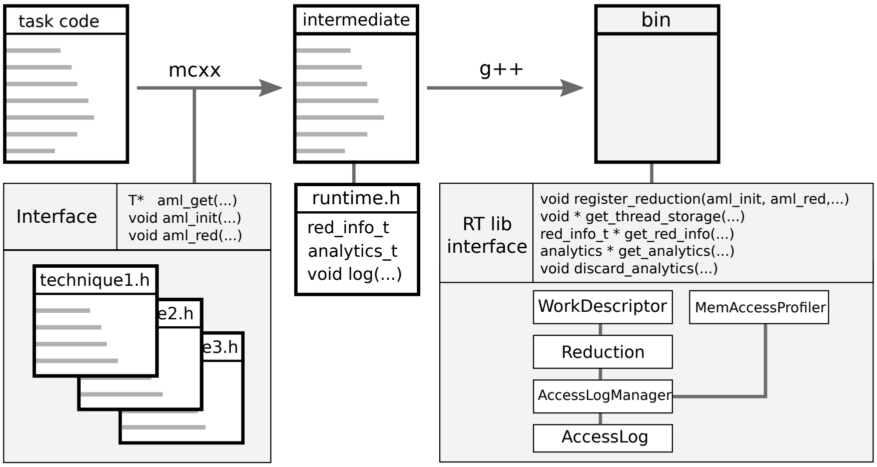


Figure 7.6: The OmpSs Reductions Model includes a set of header files that implement different techniques, a header file that implement generic data types and a runtime components handling reduction scopes and inspector-executor data.

and allocating, initializing and reducing private storages. In case of using a technique with an alternative memory layout, the runtime allocates the size of the AML type which typically is a few bytes in size and contains pointer variables. The internal allocation of a thread-private memory corresponding to a technique (such as a software cache) happens on-demand by calling the initializer during task execution. In case the runtime decides to reduce private copies, it calls the reducer function of the AML.

7.3 Towards Generalization of Reductions with AIS

Iteration ordering is a support technique for irregular array-type reductions that avoids the overheads associated with access redirection and relies on alternative iteration spaces (AIS). Alternative iteration spaces are created such that the resulting memory accesses of the loop body fall within disjoint memory regions (partition). Tasks that execute such an iteration space can be run in parallel and race condition free. The inspection of accesses to determine which iterations access which memory is an essential step that requires the use an inspector-executor.

Techniques implementing alternative iteration spaces can be grouped into two categories:

- **Privatization of Iterations.** Task- or thread-private arrays are created that hold iteration indexes that correspond to a particular memory partition. Each participating task processes one or more arrays where each array hold iterations that belong to one partition. Since partitions represent disjoint memory regions, parallel task execution is race condition free. This approach is applicable on algorithms with a static iteration space and where the ratio of size of the iteration space to array size is small. The distribution of indexes into private index arrays is based on access statistics obtained during the inspector phase.

- **Owner Computes.** This case avoids privatization of iterations. Instead each participating task iterates over the entire iteration space but executes only loop bodies of those iterations where the corresponding indexes result in an access of an owning memory location. The ownership tables are based on access statistics collected during the inspector phase. This approach works for algorithms where the majority of indexes result in conflict free updates as the opposite case would serialize the execution.

We propose a new category, *Ordering through Scheduling*, that implements alternative iteration spaces but avoids privatization of iterations nor requires tasks to process the entire iteration space. Instead, it implements race-free execution through task scheduling based on data-flows. Techniques within this category are suitable for programming models that offer support for data-flow based task scheduling. This is the case for OmpSs and OpenMP. As an example, we present a technique called *Commutative Reductions*, *ComRed* in the next section.

We are aware that a generalized support for techniques implementing iteration ordering would require a transparent support for all three aforementioned categories. This would include descriptions of APIs and code transformations. Further, altering the iteration space of an algorithm requires careful inspection of possible implications on the algorithm itself as cases exist where such transformations break the code. Since *Ordering through Scheduling* does not require to perform any code transformations and is well suited for the range of applications relevant to us, we focus on this category and defer further generalization to future work.

7.3.1 Commutative Reductions in OmpSs

ComRed builds on top of data-flow based task execution. This execution model maximizes concurrency by scheduling tasks according to memory access semantics thus avoiding manual task synchronization by the programmer. A reduction, as a sequence of read-modify-write operations, creates an *inout* dependency over the reduction variable. Since this type of dependency serializes the execution of participating reduction tasks, runtimes automatically override this dependency type by declaring reduction variables as *concurrent.*

ComRed implements a different type of dependency, namely *commutative.* Commutative dependency over a variable that permits a commutative order of execution but does not permit concurrent execution. The motivation to use this dependency type to support irregular array-type reductions is the following:

- It turns out that many applications that implement a reduction kernel exhibit a near-linear memory access pattern with small overlaps of accessed memory between tasks.
- Identifying such overlapping address regions and declaring them as commutative would allow schedulers to execute those tasks that have no overlaps.

- The identification of overlapping memory regions can be detected during an inspection phase of an inspector-executor model.

- Serialization due to overlaps can be mitigated by increasing the number of participating tasks. This ensures that enough tasks are ready to run and can make use of available resources.

Figure 7.7 shows how iterations are split into $N-1$ fine-granular tasks that are scheduled for execution in an interleaved manner on processors P_1 and P_2. This pattern is the result of commutative handling of overlapping memory regions of this technique. Such scheduling ensures that only tasks with no overlaps are running in parallel at a time.

For programming, we define a metric called *dependency stride*. It defines the minimal offset in task creation order at which any two tasks can be scheduled for execution concurrently. Figure 7.8 illustrates different dependency strides. For the case of a dependency stride of 2, any tasks can be executed in parallel with even or odd creation identifiers where the offset equals to 2. In case N tasks access the entire reduction variable *r[]*, the required dependency stride would surpass the number of available tasks and the entire execution in serialized. The Figure also shows the number of commutative dependencies created. Both, the dependency stride as well as the number of dependencies created have an impact on execution performance. We take a look at these effects in Chapter 8.

7.4 Implementing the Inspector-Executor

The inclusion of the inspector-executor into a parallel programming model requires the definition of scope and granularity and immutable task instance identification. To define a scope for an inspector, we define a region where the inspection can take place and where the application of the executor later is valid. We call this region an optimization frame. In case of a reduction, an optimization frame corresponds to the scope of a reduction computation.

Further, a definition of granularity is required on which inspectors are created and also on which inspection logging in performed (the resolution). In our case the inspector-executor is created for each individual task that participates

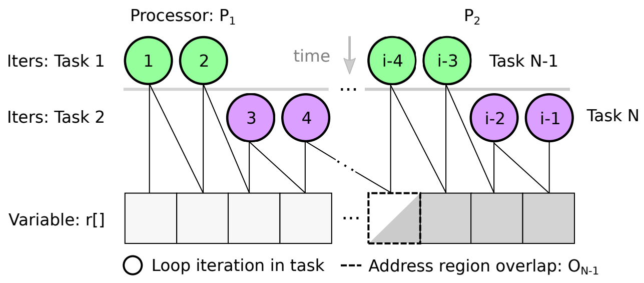


Figure 7.7: *Ordering through Scheduling* is a strategy that implements an alternative iteration space through dependency-aware task scheduling. *Com-Red* is a technique relying on commutative execution of tasks that share overlapping region(s).

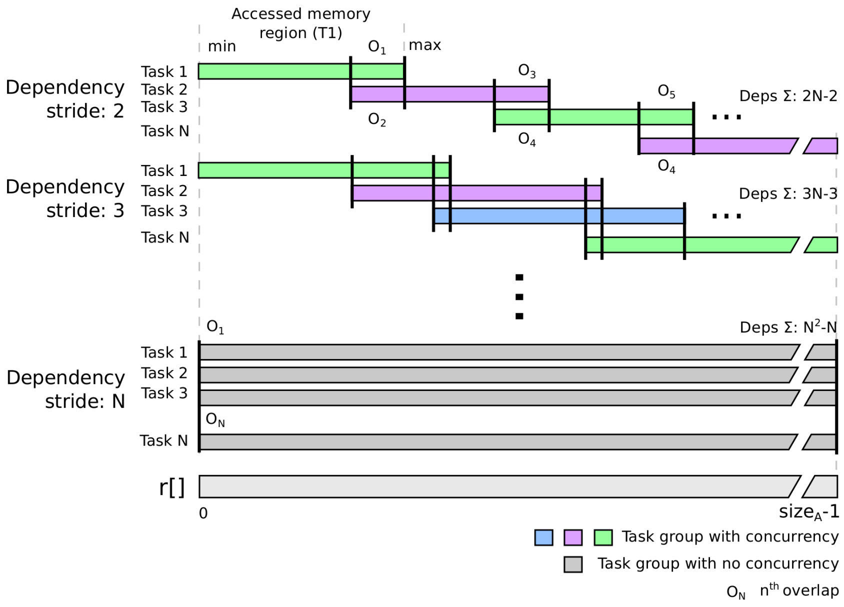


Figure 7.8: *Dependency stride* defines an offset in task creation order at which any two tasks can be scheduled for execution in parallel. In case of a scattered updated over the entire reduction array, the entire execution is serialized.

in the reduction computation. The resolution of logging is controlled by a control parameter.

Finally, since any statistical data created by the inspector for a particular task needs to be applied on the same task instance in the future during the execution phase, the runtime is required to track task instances. In case the optimization frame is identified by a hash value generated by any set of instance invariant values (such as function pointers), then particular task instances can be tracked by a simple counter that is incremented on every instantiation of a participating reduction task. This assumes that neither the creation order of tasks nor the memory access pattern of the reduction kernel change between the inspection and execution phases.

7.4.1 Implementation of the inspector

The OmpSs-RM inspector is located in a header file that is inserted into the intermediate code and implements a logger method (log). This method receives the address of the currently accessed memory location and is called from within an AML's GET-method. Every time the runtime registers a new reduction, a global inspector manager is created for that particular reduction. Participating tasks register their inspectors in the inspector manager. At the end of an optimization frame, the manager passes all logs from all inspectors to an analytics object for processing and sets a ready flag. This completes the inspection phase.

7.4.2 Code generation

Figure 7.9 shows a simplified, intermediate code where an instance-invariant identifier (*frameID*) is created to identify an optimization frame and that is subsequently passed to all participating tasks. 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 (*instanceID*). In OmpSs and for the context of reductions, the frame identifier is computed as an *xor* between the reduction target address and the value of the reducer function pointer. 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. Optimization frames across nesting levels are not supported. The intermediate task code in Figure 7.9 (line 8) shows the triplet of runtime library calls to acquire a private thread storage (*get_thread_storage*), a reduction information object containing additional information about the reduction (*get_red_info*) and access analytics (*get_analytics*). In the kernel loop itself, occurrences of the reduction target variable are replaced by a call to the GET-method as well as each access to the original reduction variable is inspected in cases where the analytics object is not ready yet.

It is important to point out that these code transformations were implemented manually in our test applications as time constraints for completing this work did not permit to add direct compiler support. Still, code organization and transformation steps were designed with feasibility for automation of this process in mind.

```
 1 while(...){
 2   frameID = instance_invariant_identifier;
 3   handle_optimization_frame(ID) //compute anltcs
 4    task = new reduction_task(frameID,taskcode,...)
 5    task.run();
 6 }
 7 ...
 8 taskcode(...){
 9   void * v_priv; red_info_t * i; analytics_t * a;
10   v_priv = get_thread_storage(v);
11   i      = get_red_info(v);
12   a      = get_analytics(v, a->instanceID);
13   for(...) {
14     j = f(j);
15     if(!a->ready)
16         inspect(&v[j],v  , a, a->frameInstanceID);
17     (*aml_get(&v[j], v_priv, i, a))++;
18   }
19 }
```

Figure 7.9: A simplified, intermediate code shows the support of AMLs and inspector-executors in OmpSs

7.4.3 Redirection with Selective Privatization

The availability of an inspector-executor in parallel programming models allows support for selective privatization. It represents an optimization technique where redirection of accesses into thread-private containers can be avoided if the accessed memory address is within the range of an owned memory region. By inspecting accesses during the inspection phase, an inspector can generate knowledge on which tasks access which memory regions. A task that accesses a particular memory region with the highest frequency becomes the owner of that region. Later, during the execution phase, this task can avoid redirection and access the owning memory region directly. In case multiple tasks have an equal number of accesses to a region, a heuristic can be used to determine the ownership. All accesses into non-owning memory are redirected into a thread-private storage container (AML). In order to avoid data races between ongoing direct accesses and the reduction of private storages during computation, an intermediate copy is used to keep data apart. We call this intermediate reduction array copy as *Recombination Storage.*

Selective privatization is generally applicable on approaches that implement access redirection. A schematic representation of this optimization technique is shown in Figure 7.10. In this example, the inspection granularity equals to the number of elements which allows to define ownership for individual array elements. Since index four results in an access to a memory region owned by another task, is redirected into a thread-private container.

In our work we implement selective privatization using access redirection with PIBOR.

7.5 Language Support for AMLs and AIS

To program with OmpSs-RM, we require three additional pieces of 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 propose the extension of the reduction clause by the additional parameter $MODE$, where mode is an identifier of a vendor provided privatization technique. We define a reduction clause using the mode identifier as

$$reduction\ (id : target : mode)$$

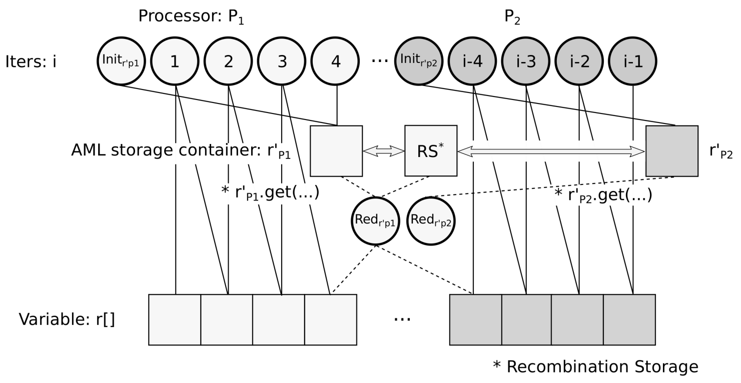


Figure 7.10: *Redirection with Selective Privatization* combines benefits of an inspector-executor approach with access redirection into thread-private data containers. This allows to bypass access redirection for accesses into owned memory thus effectively reducing the overhead of access redirections.

where *reduction* corresponds to the OpenMP reduction clause , *id* is an operator and *target* a variable identifier.

Further, in order to enable techniques 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. It is defined as

$$reduction\ (id : target : mode)\ invariant(target)$$

Lastly we propose the addition of the *loopstep* pragma. This pragma declares the encountering region as an optimization frame and is used to differentiate between inspection and execution phases. It is defined as

$$\#pragma\ omp\ loopstep$$

```
/*region*/
while(simulation_runs()){
#pragma omp parallel for reduction (+:v:SP) invariant (v)
  for(int i = START; i < END; i++) {
   j = f(i);
   v[j]++;
  }/*end of reduction*/
  #pragma omp loopstep
}/*end of optimization frame for v*/
```

Figure 7.11: OmpSs implementation of a reduction kernel showing the proposed language extensions

On encountering a loopstep pragma, the runtime may transition from executing an instrumented code (inspector) to an optimized code version. The pragma applies to all targets that were declared in the invariant clause in the same region. Figure 7.11 shows an example of an array reduction using the proposed language extensions where MODE is set to selective privatization (*SP*).

7.5.1 Restriction

A variable that is declared as a reduction target with a MODE set to an AML may not be passed to an external function or library due to mismatching memory layouts.

7.6 Conclusion

In this Chapter we presented the OmpSs Reduction Model. It extends the OmpSs programing model by language and runtime features to support a programmable and scalable execution of array-type reductions. This support allows the use of techniques that are based on the underlying optimization strategy of access redirection and iteration ordering.

Access redirection can be generalized by allowing the use of custom thread-private data containers and associated initializer-, reducer- and data placement functions. Techniques implementing such a data container and access functions are called techniques with alternative memory layouts (AMLs). Examples for this category are CachedPrivate, PIBOR and PIBOR with Selective

Privatization. Since a data placement function can improve data locality, these techniques are typically suited for irregular array-type reductions.

Further, to support iteration ordering we presented a technique called Commutative Reductions and a new category named Ordering through Scheduling. Commutative Reductions use the underlying mechanics of dependency tracking of task data to implement a race condition-free correct scheduling. Declaring overlapping memory regions as commutative allows the scheduler to run only those tasks in parallel that result in conflict-free task execution. This technique works well for near-regular reductions where access overlaps between tasks are small.

Both, Commutative Reductions and Selective Privatization require the use of an inspector-executor. In this Chapter we presented an implementation of this execution model in OmpSs and discussed the concepts of optimization frame as well as the required language and runtime features.

In the next Chapter we present case studies and discuss performance results of the aforementioned techniques.

CHAPTER 8

Case Studies

Our work on support for array-type reductions was motivated by LULESH [2] and SPECFEM3D [5]. Both applications represent numerical algorithms with data motion and programming style typical for scientific computing. Their underlying discretization with irregular meshes as well the use of nodal and element-wise centering (staggered mesh) to store different values suggest an irregular memory access pattern. However, specific problem sets and the use of techniques such as mesh coloring reduce the degree of irregularity resulting in near-linear access patterns. This property makes both applications interesting to evaluate techniques such as *PIBOR with Selective Privatization* and *Commutative Reductions*. They promise the highest payoff for such cases.

In this Chapter we provide more information on both applications and discuss results obtained with different techniques and hardware architectures. Results show programmability and scalability of our solution.

8.1 Introduction

In the previous chapters we stated that techniques with access redirection into thread-private containers improve data locality for irregular reductions if a suitable data placement function is provided. *PIBOR* is an example hereof. Its placement function redirects accesses into thread-private bins corresponding to memory regions of the reduction array. This technique achieves the highest speed-ups for single and multi-threaded executions of the *RandomAccess* micro application.

We also claimed that techniques with iteration ordering can be applied in cases where the cost of ordering remains small. As an example we presented a technique called *Commutative Reductions*, *ComRed*. This technique implements an alternative iteration space through dependency-aware task scheduling. This technique avoids disadvantages associated with *Privatization of Iterations* or *Owner Computers* as it implements a more efficient way of ordering, namely through task scheduling. However, it requires an inspector to identify memory overlaps. The cost of inspection, potential for amortization as well as performance scalability of this approach remains unclear so far.

Therefore, in this Chapter, we iterate over all support techniques including ComRed and apply them on *LULESH* and *SPECFEM3D*. We consider the following topics to be of interest:

- Memory access patterns of the reduction kernels
- Impact of atomics and replication
- Impact of techniques with redirection into an AML and selective privatization (PIBOR)
- Impact of technique with iteration ordering (Commutative Reductions)
- Cost of inspection
- Scalability across threads and nodes

In the following sections we provide more information on the applications and systems used and present performance results. We also discuss the aforementioned points of interest.

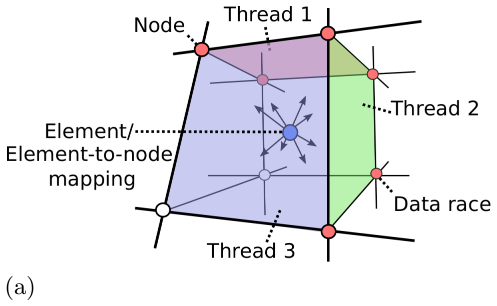


Figure 8.1: Element-centric computation over a staggered mesh results in scatter-updates to nodal variables once the computation of values finishes. This represents the irregular array-type reduction in the LULESH hydrodynamics code which simulates a wave blast propagation in a compressible medium (*b*)

8.1.1 Applications

LULESH: The *Livermore Unstructured Lagrange Explicit Shock Hydrodynamics*, is a simplified, Lagrangian, inviscid compressible hydrodynamics code adopted from the ALE3D application suite. It simulates the distribution of a blast wave through a homogeneous medium in 3D.
The simulation domain is discretized in space with a staggered mesh and hexahedral mesh elements. Staggered mashes hold variables at node- and element-level which are typically stored in different arrays. Element-wise processing includes three steps in each iteration: a gather of data that corresponds to nodes of an element, applying different numerical methods that update element- and node-variables and making changes persistent by updating node variables. This behavior can be observed in the reduction kernels *IntegrateStressForElems()*, shown in Figure 8.2, and *CalcFBHourglassForceForElems* which is similar in structure.

In this example, the method *CollectDomainNodesToElemNodes()* aggregates nodal values into temporal variables *f{x,y,z}_local*. These aggregated values are used later in computation but with element-wise centering. Once the computation completes, nodal values are applied back to the respective nodes. This represents the typical update operations in a reduction kernel as found in many related applications. The relationship between elements and nodes is stored in the *nodelist* array. A spatial distribution of variables in a mesh element is shown in Figure 8.1(a). LULESH is ongoing development and is currently available in version 2 [31].

SPECFEM3D: SPECFEM3D simulates a seismic wave propagation at the local or regional scale based on the spectral-element method (SEM). It implements a reduction kernel which is similar in composition and in runtime behavior as seen in LULESH. The reduction kernel is shown in Figure 7.2.

SmartJumper: SmartJumper[6] is a parametrized and templated C++ implementation of the popular RandomAccess micro benchmark[7]. SmartJumper consists of a kernel method that implements an array-type reduction, a command line parser and a correctness verification code. Performance behavior is influences by array-size, number of updates, the order of accesses (sequence), access distribution and degree of accessed memory overlaps between processors (locality). The sequence and frequency at which array elements are accessed (distribution) are configured through template parameters. SmartJumper provides abstract classes for both, generators and distributions. We have implemented a linear and random number generator. Distributions include a linear and Gauss-distribution.

8.1.2 System Configuration

Benchmark results were obtained on the MareNostrum3 supercomputer equipped with a 2-way Intel Xeon E5-2670 CPU with eight cores each and a 4-way IBM POWER8 system equipped with 6 cores each and eight native treads per core (SMT). On the POWER8 system, a thread striding offset of eight was used in order to distribute threads across cores first.

Applications were compiled with the OmpSs Mercurium compiler v1.99 and GCC v5.1.0 or Intel C Compiler 15.0.2 as native compilers with -O3. The runtime is based on the Nanos++ RTL v0.9a.

8.2 Performance Evaluation of LULESH

In Figure 8.4 we show scalability results of both reduction kernels of the LULESH application obtained on a Xeon E5 SMP node. We have set the

```
//chunking and offsetting code
for(i=0;i<num_tasks;i++){
  //chunks assignement code here
   #pragma omp task in(domain.m_x, domain.m_y, \
                     domain.m_z, domain.m_nodelist,\
                     *sigxx, *sigyy, *sigzz)\
                      concurrent(*determ)\
                      firstprivate(kk, upper_limit ) \
                       label(IntegrateStressForElems) \
                      reduction(+:fx:COMRED)\
                      reduction(+:fy:COMRED)\
                      reduction(+:fz:COMRED)\
                      invariant(fx, fy, fz)
  for( Index_t k=kk ; k<upper_limit ; ++k ) {
    const Index_t* const elemToNode = domain.nodelist(k);
    Real_t fx_local[8] ;
    Real_t fy_local[8] ;
    Real_t fz_local[8] ;

    CollectDomainNodesToElemNodes(fx_local,fy_local,\
                                              fz_local,...);
    Computation1(fx_local,fy_local,fz_local,...);
    ...
    ComputationN(fx_local,fy_local,fz_local,...);

    for( Index_t lnode=0 ; lnode<8 ; ++lnode ) {
      Index_t gnode = elemToNode[lnode];
      fx[gnode] += fx_local[lnode];
      fy[gnode] += fy_local[lnode];
      fz[gnode] += fz_local[lnode];
    }
  }
}
#pragma omp loopstep
```

Figure 8.2: IntegrateStressForElems() with explicit tasking and the proposed language extensions. This method shows how node values are gathered, processed and distributed back, thus creating the scattered update access pattern typical for array-type reductions.

```
void SmartJumper<T>::kernel(int taskNum, SIZE_T iters, SIZE_T block,\
ARRAY_T * a)
{
  SIZE_T a_size = settings->numArrayElements;
  #pragma omp task label(kernel) reduction(^:[a_size]a:PIB)
  {
    Generator<T> gen(taskNum * iters, taskNum, settings);
    for(auto i = 0; i < iters; ++i){
      SIZE_T pos = (*distro)(&gen);
      a[pos] ^= pos;
    }
  }
}
```

Figure 8.3: SmartJumper is a modern implementation of the popular RandomAccess benchmark

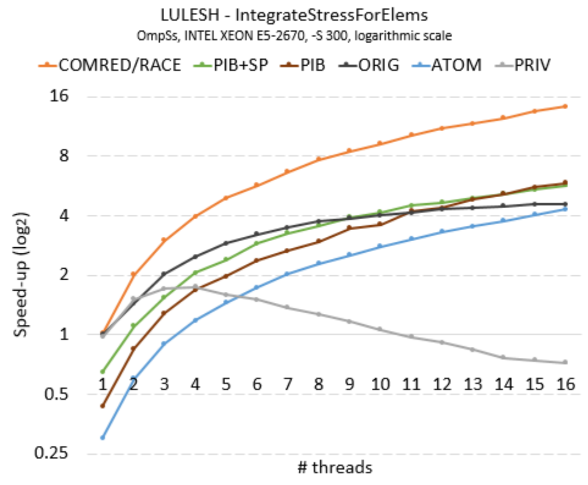
problem size to 300 elements per dimension which results in acceptably short execution times and a reduction array of 200MB. The time information used corresponds to the execution of one iteration of the corresponding reduction kernel.

The graph shows speedups on a logarithmic scale relative to serial execution time over different thread counts and with chart lines corresponding to different techniques. Looking at the 16-thread execution, the highest speedups are achieved with ComRed where the performance results are equal to the hypothetical implementation named (*RACE*). The inspector in this case was set to a resolution of 128 regions which corresponds to region sizes of approximately 1.5MB.

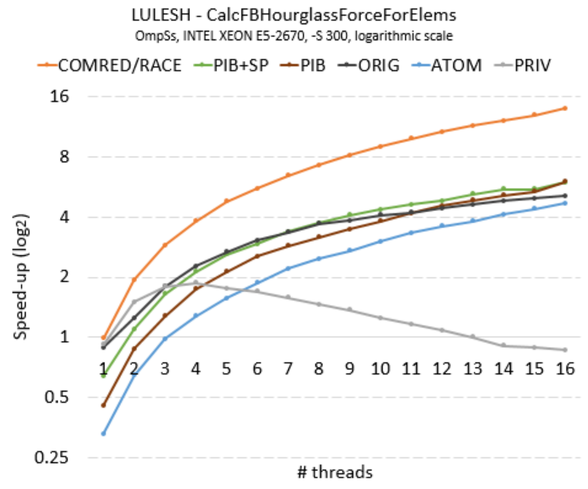
RACE is an implementation that contains data races but avoids overheads that occur when using support techniques. This version represents a hypothetical upper scalability bound.

Atomics and an implementation called *ORIG*, followed by *PIBOR* and *PIBOR with Selective Privatization* achieve lower speedups. The worst scalability is obtained when using privatization with array replication.

ORIG represents the original, concurrent implementation of LULESH. This version follows the idea of privatization with array replication. However, instead of creating copies for each thread, the reduction array is expanded by

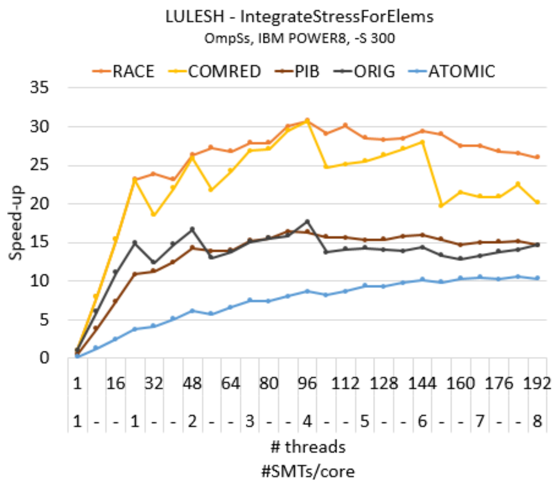


(a)

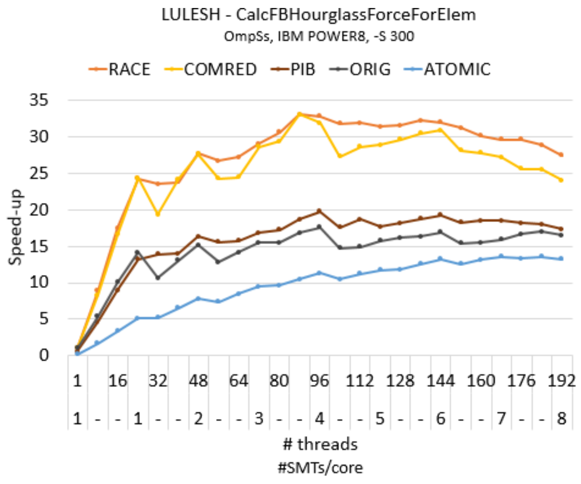


(b)

Figure 8.4: Performance scalability of both LULESH reduction kernels obtained on a 2-way Intel Xeon E5 processor with 16 cores shows that *Commutative Reductions* achieves a four times higher speedup compared to other techniques and is *en par* with the unprotected version marked as *RACE*.



(a)



(b)

Figure 8.5: Performance scalability of both LULESH reduction kernels reaches a plateau with 96 threads and 4 SMT threads per core. Also in this case, *Commutative Reductions* produces the highest speedups compared to other techniques.

a factor of eight. This number corresponds to the number of nodes of each indexed element (shown in Figure 8.1(a). Since processing each index results in updates of eight disjoint array positions, this approach is race condition free but comes at the cost of allocating, initializing and reducing eight array copies.

It is important to point out that scalability results in the presented charts exclude the first iteration of the algorithm. A performance chart including the inspection phase would depend on the number of consecutive iterations that amortize the overhead of the inspection phase. We take a more detailed look in the following sections.

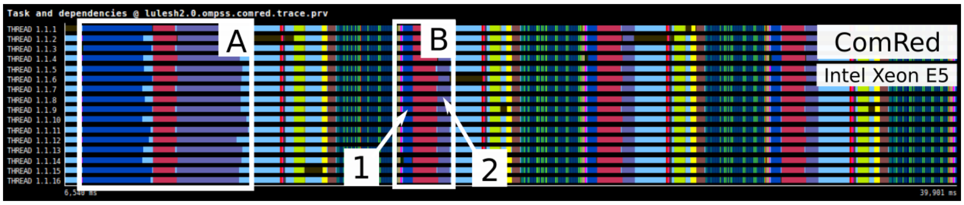
Figure 8.5 shows performance scalability of the LULESH reduction kernels with different techniques obtained from the IBM POWER8 system. To balance loads across cores we used a thread placement offset of eight. This offset distributes threads to cores first and avoids initial hot-spots of high thread counts per core. Consequently, with each 24 threads that are being added to the benchmark run, the number of SMT threads per core increases by one. The relation between SMT threads and cores is shown in the horizontal axes of the chart. Since executions with smaller SMT counts per core reduce resource sharing between threads, they achieve the highest performance results (peaks).

8.2.1 Discussion

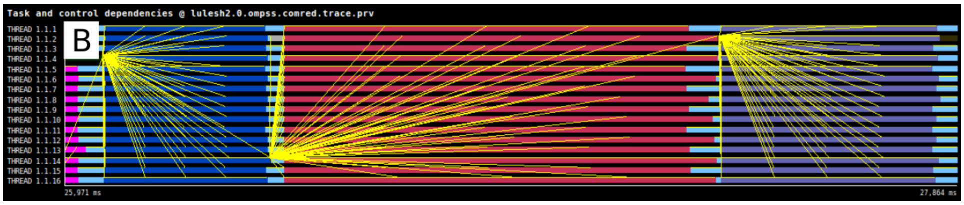
The presented scalability results obtained from the Intel Xeon E5 and IBM POWER8 system bring up a set of interesting questions that we would like to discuss in this section.

Why is ComRed performing best? The LULESH reduction kernels manifest a near-regular access pattern and small overlaps between owned memory regions. This is due to the linear relationship between elements and surrounding nodes and geometric properties of the mesh. Commutative Reductions achieves the highest speedups in this case as overheads of data replication or atomic updates can be avoided. Instead, this technique relies on task scheduling only.

Figure 8.6 shows the execution trace on the Xeon E5 system implemented with Commutative Reductions. The repetitive pattern reflects the iterative nature



(a)

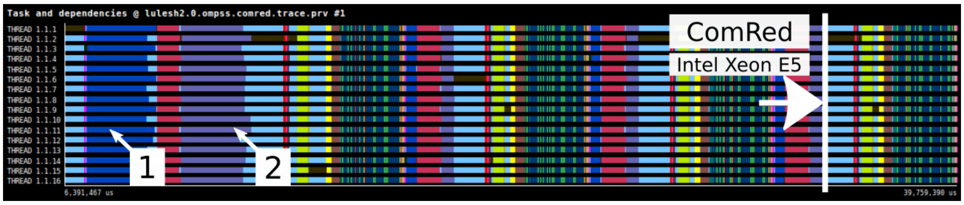


(b)

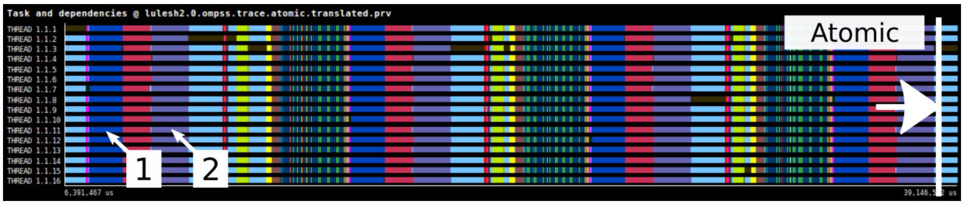
Figure 8.6: Execution trace of LULESH with Commutative Reductions shows the first four iterations of the global global simulation loop. The augmented view B (*b*) of the marked region shows three taskified methods including the two reduction kernels (*1* and *2*) with the associated dependencies (lines).

of such simulation codes due to the global simulation loop. Taking a closer look at one iteration (area *B*) shows the sequence of the two reduction kernels *IntegrateStressForElems*, (*1*) and *CalcFBHourglassForceforElems*, (*2*). It can be seen that the inspection phases (area *A*) for both reduction kernels take significantly more time than the execution phases. However, once inspection is completed, execution phases perform *en par* with the hypothetical version described as *RACE*. In this trace, horizontal bars depict threads and colors mark tasks that correspond to different methods. Yellow lines represent data dependencies between tasks excluding commutative dependencies.

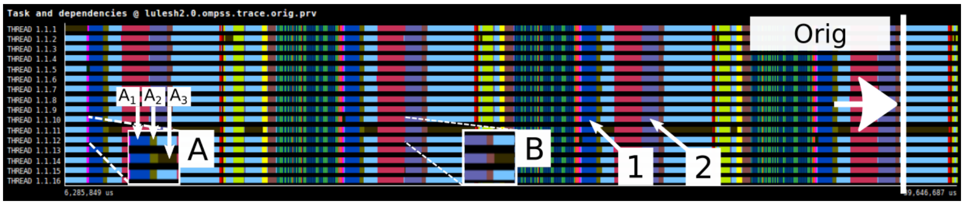
Figure 8.7 shows the execution trace of ComRed in comparison to executions with atomics and the original implementation using array expansion. The number identifiers *1* and *2* correspond to the two reduction kernels in the application. The traces show that the cost of inspection is amortized after the fourth iteration and gives insight into different overheads associated with each technique.



(a)

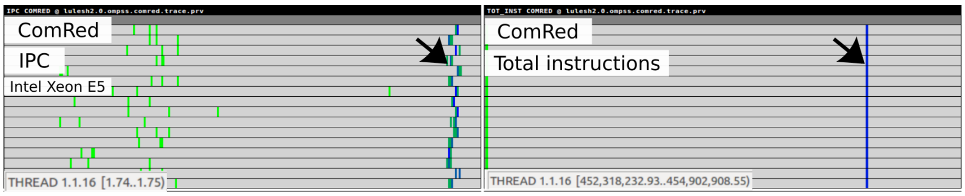

Task and dependencies @ lulesh2.0.ompss.trace.atomic.translated.prv
THREAD 1.1.1
THREAD 1.1.16
6,391,467 us
Atomic
1
2

(b)

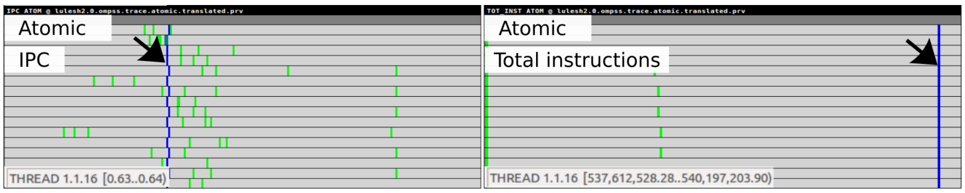


(c)

Figure 8.7: Execution traces show differences in execution speeds and time gains after the fourth iteration of the LULESH simulation loop with both reduction kernels *1* and *2* implemented with different techniques.



(a)

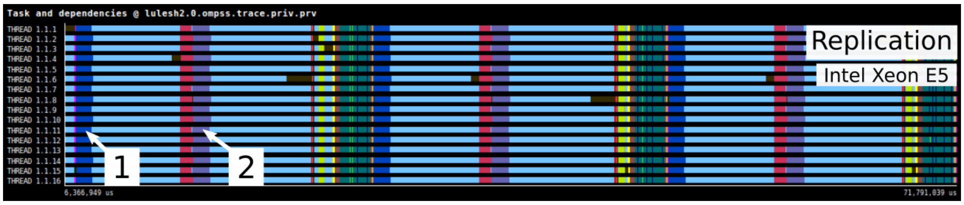


(b)

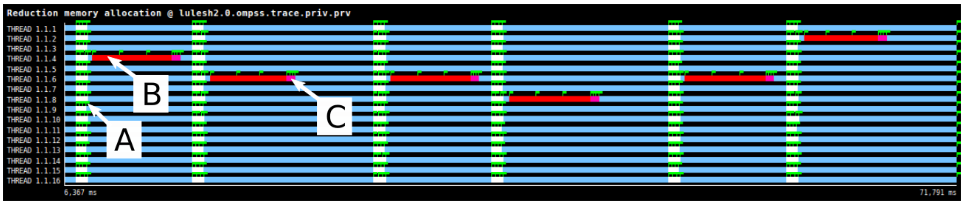
Figure 8.8: Histograms over IPC and total instructions show that the use of atomics in the LULESH reduction kernel causes a drop in IPC by a factor of 2.8 and increases the total instructions count by 18 %. This results in a performance degradation by a factor of approximately 4 compared to the implementation using ComRed.

For ComRed, the overhead associated with the inspection phase dominates the first iterations. The cost of inspection is composed of cost of technique and cost of creating address histograms and address ranges for each reduction variable and task. This statistical information is used to compute ownerships and to define memory overlaps which are used later for commutative scheduling. Since during inspection a commutative execution of tasks is not applicable yet, a technique is needed to avoid data races during that phase. In our case we use PIBOR (redirection into bins).

The use of atomics introduces additional instructions and decreases the rate at which instructions are executed per cycle (IPC). This decrease in IPC is related to the processor specific implementation of atomicity. Figure 8.8 shows a histogram of instructions and instructions per cycle of the *IntegrateStressForElems* reduction kernel on the Intel system. It shows that the use of atomics increases the total number of instructions by 18 % while the IPC rate degrades by a factor of 2.8.



(a)


Reduction memory allocation @ lulesh2.0.ompss.trace.priv.prv
THREAD 1.1.1
THREAD 1.1.16
A
B
C
6,367 ms
71,791 ms

(b)

Figure 8.9: LULESH with replication does not scale due to large overheads associated with the initialization (A), reduction (B) and deallocation (C) of thread-private array copies.

The version with array expansion yields a lower IPC during task execution compared to ComRed and requires an additional reduction phase of data stored in the expanded array. Each kernel is then followed by a *free* system call. This sequence is shown in 8.7(c) in regions A and B corresponding to the two reduction kernels, where regular task execution (A_1) is followed by a concurrent reduction phase (A_2) and single threaded activity that corresponds to freeing memory (A_3).

Array privatization with data replication that relies on data replicas is not scalable due to private memory handling. The IPC during task execution is below the version of array expansion (0.9 vs. 1.1) but it is the concurrent initialization and serial reduction of thread-private copies once the kernel finishes which affects performance most. Figure 8.9 shows the sequence of allocation, initialization and serial reduction marked as A, B and C. Each phase shows three event flags (green markers) that mark the event generation for each of the three reduction targets *fx*, *fy* and *fz* as shown in the code in Figure 8.2.

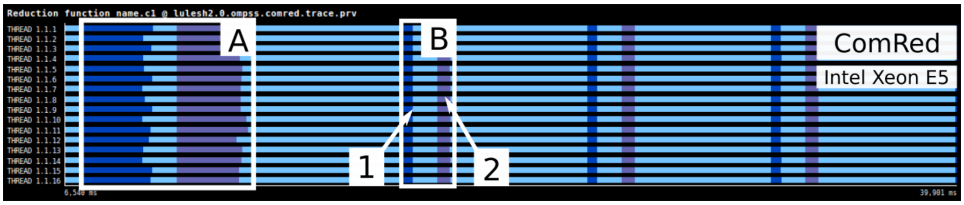
When does ComRed underperform? Commutative Reductions is a technique that underperforms in the following cases:

- Memory access pattern is unsuitable. Irregular memory accesses with tasks accessing the entire or large portions of the reduction array cause a serialization of participating tasks. This case is trivial and does not require further investigation.
- Not enough tasks are created. Any two tasks with overlapping memory accesses are serialized. In this case it is mandatory to inspect the dependency stride length and to create a multiple of tasks to obtain concurrency.
- Creation order does not allow to schedule the immediate successor for execution. This is a hypothetical limitation as the performance impact depends on the actual runtime implementation.

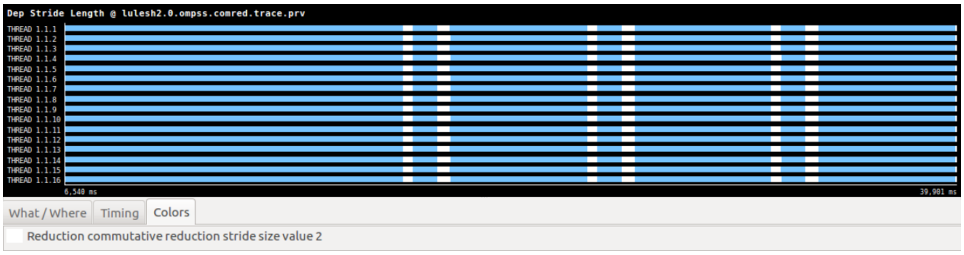
Creating a sufficiently large number of tasks is a requirement to achieve concurrency and scalability with ComRed in LULESH. Taking a look at both reduction kernels shows that the dependency stride length is 2. As defined in the previous Chapter, the dependency stride length is the minimal offset in task creation order that guarantees concurrent execution. Consequently LULESH requires at least twice as many tasks as available threads in order to generate enough interdependent work to achieve a 100% thread utilization. Figure 8.10 shows an execution trace obtained from the Intel Xeon system showing only reduction tasks and an overlay (8.10(b)) that allows programmers to see the computed dependency stride length.

Figure 8.11 shows two executions of LULESH with different number of tasks. It can be seen that in case of creating 16 tasks only, half of the execution resources remain idle (8.11(b)) as not enough tasks are available for concurrent execution.

The task dependency stride length has another important implication. It turns out that performance is significantly impacted if the task creation order is such that the immediate successor cannot be scheduled for execution immediately. This is the case in common code implementations when the task generating loop of the reduction kernel continuously creates tasks with


Reduction function name.c1 @ lulesh2.0.ompss.comred.trace.prv
THREAD 1.1.1 … THREAD 1.1.16
A
B
1
2
ComRed
Intel Xeon E5
6,540 ms
39,901 ms

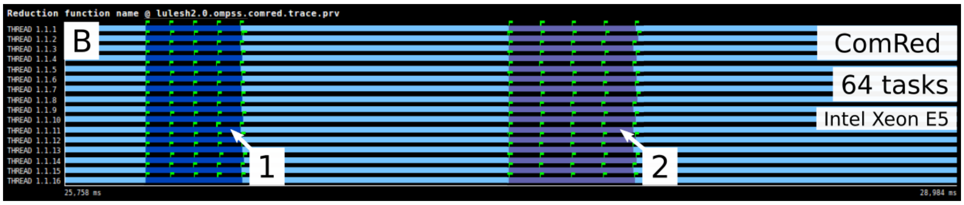
(a)


Dep Stride Length @ lulesh2.0.ompss.comred.trace.prv
THREAD 1.1.1 … THREAD 1.1.16
6,540 ms
39,901 ms
What / Where
Timing
Colors

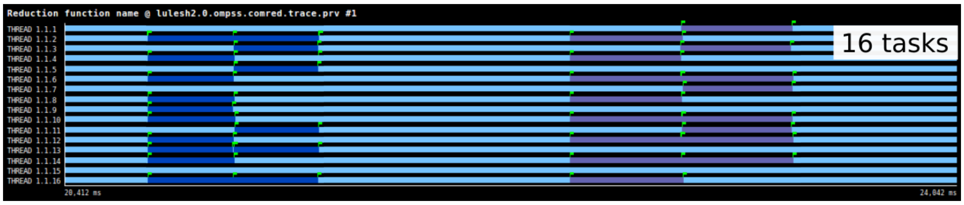
Reduction commutative reduction stride size value 2

(b)

Figure 8.10: Execution traces of LULESH showing only reduction tasks and the dependency stride length of 2 (b) computed during the inspection phase.


Reduction function name @ lulesh2.0.ompss.comred.trace.prv
THREAD 1.1.1 … THREAD 1.1.16
B
1
2
ComRed
64 tasks
Intel Xeon E5
25,758 ms
28,984 ms

(a)


Reduction function name @ lulesh2.0.ompss.comred.trace.prv #1
THREAD 1.1.1 … THREAD 1.1.16
16 tasks
20,412 ms
24,042 ms

(b)

Figure 8.11: A dependency stride length of 2 requires at least twice as many tasks as available threads in order to generate enough independent work to keep all threads busy.

consecutive iteration chunks resulting in memory overlaps always between the current and the successor task.

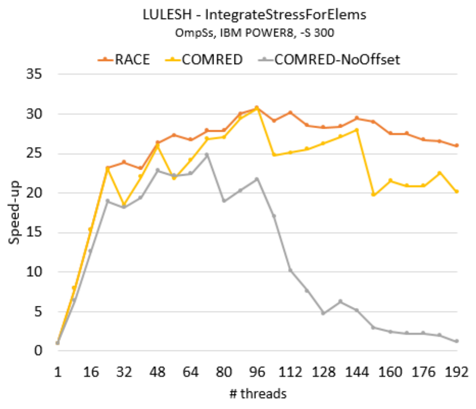
Since the successor task has a commutative access defined over the overlapping region, it cannot be scheduled for execution and is queued into the task ready queue. Once a task is placed into the ready queue all idle threads begin to poll for work. Each polling activity requires to check if any other task with the matching commutative memory region is currently in execution. Is that the case, polling continues. In the opposite case, the task is removed from the queue and lock variable is set to indicate that a task with the particular commutative access is currently in execution. The activity pattern results in high lock contention that also leads to decreased performance of the task generating thread.

A solution can be achieved in two ways. One possible solution is an improvement of the implementation of commutative in the runtime by substituting the polling semantic by a notification semantic. The other solution is to change the task creation order implemented by the task generating loop of the reduction kernel such that consecutive tasks are free of overlaps. Following the visualization scheme in Figure 7.8, this corresponds to the order where tasks are created group by group and where each group corresponds to one color. This solution can be compiler supported.

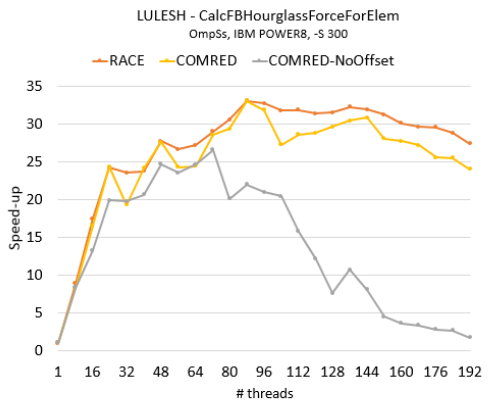
Figure 8.12 shows the extent of performance degradation when task creation order does not take the mutual exclusive execution of consecutive tasks into account (shown in chart line *COMRED-NoOffset*). In our experiments, we use a task creation offset of 2 (*COMRED*).

Figure 8.13 shows two execution traces of LULESH on the POWER8 system running with 96 threads. In Figure 8.13(a), the task creation offset of 2 is used which allows the immediate execution of the successor task (marked by the same color). The lower view in this Figure shows the relativly small amount of polling activity by worker threads. Figure 8.13(b) shows the execution trace where no task creation offset is used and where each successor is placed in the ready queue first. The amount of polling is high.

What causes performance peaks on the POWER8 system with Commutative Reductions? Previously we have described that ComRed



(a)

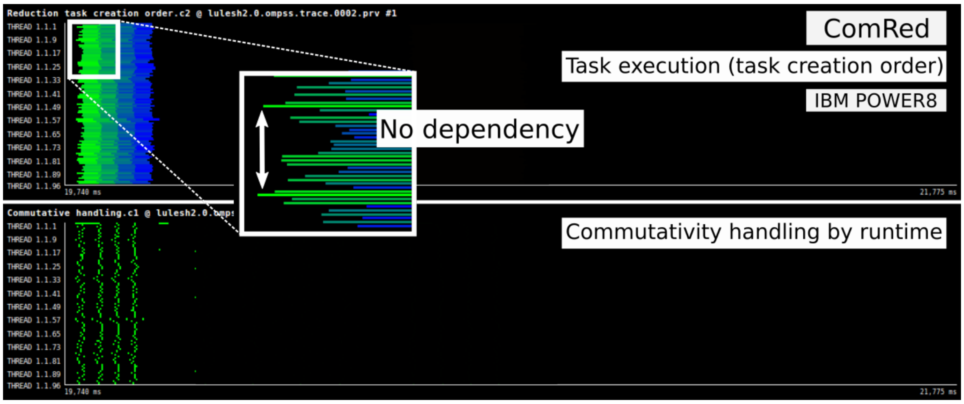


(b)

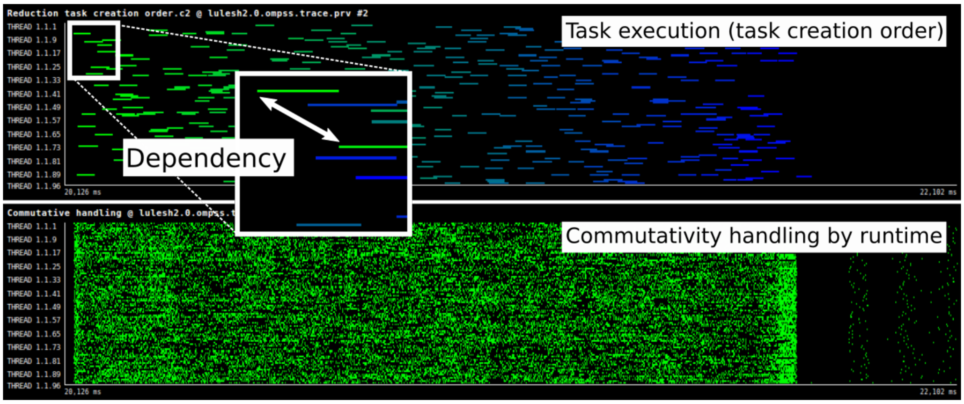
Figure 8.12: ComRed relies on an efficient implementation of commutative task scheduling or on a suitable task creation order that allows to generate independent task sufficiently fast. The opposite case leads to performance degradation for larger thread count which is shown in the graph as *COMRED-NoOffset*.

requires a sufficiently large number of ready tasks and that the creation order has an important contribution to that. A suitable creation order which allows the immediate execution of the successor task reduces polling and allows tasks with no commutative dependency to run in parallel. Unfortunately scenarios occur where the creation and execution order is disturbed due to imbalances in execution speed. For LULESH and the IBM POWER8 system, such imbalances are introduced by irregular distributions of SMT threads on the system. Processor cores with more SMT threads lag in execution speed for this particular application due to resource sharing. As a consequence, tasks running on slower threads do not finish on time and start blocking other tasks from execution.

Figure 8.14 shows execution traces of one reduction kernel of LULESH created on the IBM POWER8 system with 96 and 102 threads. Executions with 96 threads can maintain the creation order which can be seen by the color blocks. This changes for excutions with 102 threads. In this case the first processor has one additional SMT thead per core.


Reduction task creation order.c2 @ lulesh2.0.ompss.trace.0002.prv #1
ComRed
Task execution (task creation order)
IBM POWER8
No dependency
Commutative handling.c1 @ lulesh2.0.ompss
Commutativity handling by runtime
19,740 ms
21,775 ms

(a) Task creation order with offset of 2


Reduction task creation order.c2 @ lulesh2.0.ompss.trace.prv #2
Task execution (task creation order)
Dependency
Commutative handling @ lulesh2.0.ompss.t
Commutativity handling by runtime
20,126 ms
22,102 ms

(b) Original task creation order creates a dependency between current task and its immediate successor

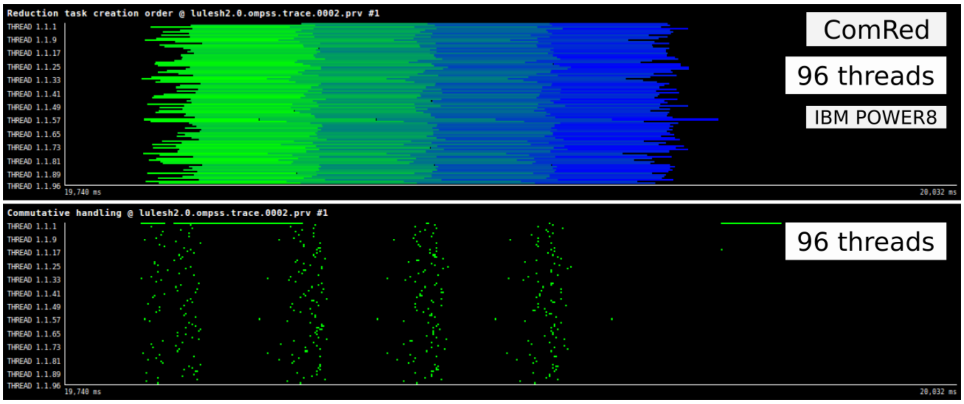
Figure 8.13: Creating tasks in a sequence where the immediate successor has a commutative dependency on the previous task results in a high lock contention, slows down task creation and degrades the number of ready tasks.

The resulting differences in IPC (instructions per cycle) are shown in Figure 8.15. The upper trace shows histograms of overall IPC, IPC of the reduction kernel and IPC overlaying the thread view for 96 threads. The vertical axes represent thread numbers ranging from 1 to 96. Color blocks at the beginning of the trace mark NUMA nodes. Since we use a thread distribution offset on this NUMA system of 8 (size of SMT threads per core), the color changes every 6 threads (number of cores per processor). The lower execution trace illustrates the unbalanced case with 102 threads. This thread count adds an additional colored box since 6 threads are added to the first processor. In this configuration, one processor runs 5 SMT threads per core, whereas the other 3 processors run 4 SMT threads per core. It can be seen that the IPC rate drops on the processor running 5 SMTs per core, visualized as 5 groups of 6 SMT threads each. The resulting performance difference corresponds the performance drop as seen in the scalability chart in Figure 8.5

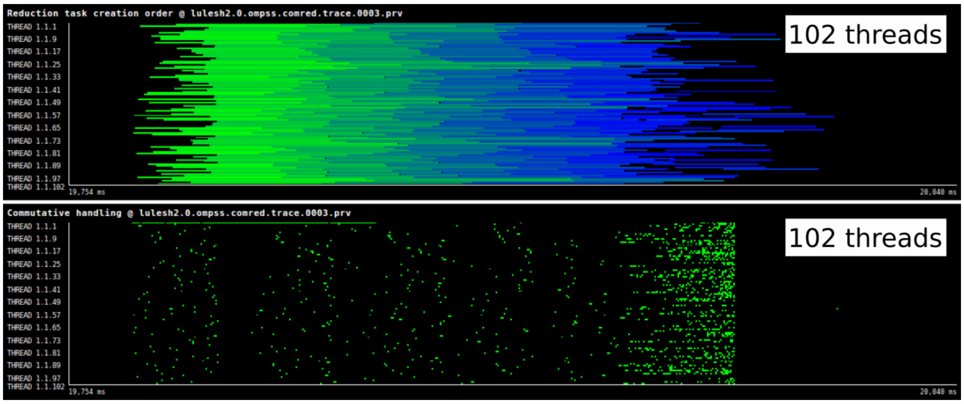
Why does Selective Privatization not achieve a higher scalability? *Selective Privatization* promises to reduce the amount of redirections by giving exclusive access to memory regions to tasks with the highest access frequency to that particular memory region. Such a task becomes the owner of that region. Interestingly it turns out that for LULESH, selective privatization is not particularly beneficial. While some tasks avoid redirection and thus achieve higher execution speeds on the executing thread, other tasks redirect more accesses into thread-private data containers thus slowing down the overall execution of the kernel. We call the percentage of owned accesses to total accesses as hit rate.

Since memory regions being used to define ownerships do not correspond to the exact starting addresses and sizes of the memory overlaps, the number of tasks that yield high hit rates differ per execution. Further, the requirement to reduce the recombination buffer at the end of execution reduces overall execution performance of the reduction kernel.

Interestingly, increasing the number of tasks does not improve hardware utilization as more tasks result in more region overlaps and consequently in an higher total percentage of privatized accesses.


Reduction task creation order @ lulesh2.0.ompss.trace.0002.prv #1
ComRed
96 threads
IBM POWER8
19,740 ms
20,032 ms
Commutative handling @ lulesh2.0.ompss.trace.0002.prv #1
96 threads

(a) Execution with 96 threads


Reduction task creation order @ lulesh2.0.ompss.comred.trace.0003.prv
102 threads
19,754 ms
20,040 ms
Commutative handling @ lulesh2.0.ompss.comred.trace.0003.prv
102 threads

(b) Unbalanced execution with 104 threads

Figure 8.14: Certain thread counts result in nonuniform distribution of SMT threads over available cores. The resulting execution speed imbalance reduces the efficiency of task creation offsetting especially towards the end of the execution of the reduction kernel. This effect is visible in the execution traces of the *IntegrateStressForElems* method on the IBM POWER8 system when comparing different thread configurations.

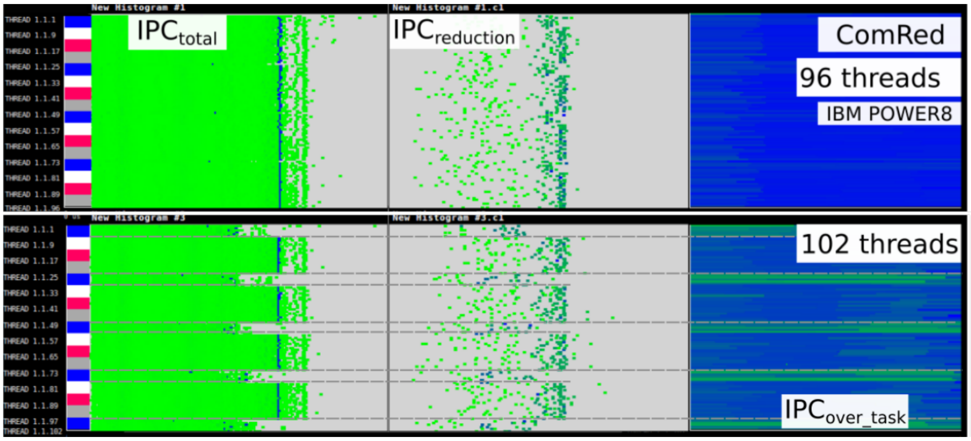


Figure 8.15: Unevenly distributed SMT threads shown in Figure 8.14 result in execution speed differences and manifest as lower IPC rates on the affected processor cores (lower picture). Differences in execution speed affect the efficiency of the ComRed technique.

Figure 8.16 shows the hit rates achieved with selective privatization and the PIBOR technique applied to the *IntegrateStressForElems()* reduction kernel in LULESH.

What are the performance benefits on clusters? Commutative reductions work similarly in cluster environments. In Figure 8.17 we compare MPI versions of both reduction kernels of LULESH (*1* and *2*) implemented with OmpSs and OpenMP. The OpenMP version represents the original source code using array expansion by 8. The trace shows the additional time spent in the reduction (marked as region A) and freeing (marked as region B) of the memory copies. Since performance and scalability behavior are comparable to executions on shared-memory systems, we do not present further analysis.

8.3 Performance Evaluation of SPECFEM3D

SPECFEM3D is an example of a near-linear array-type reduction with small overlaps between accesses produced by the participating tasks. Figure 8.18

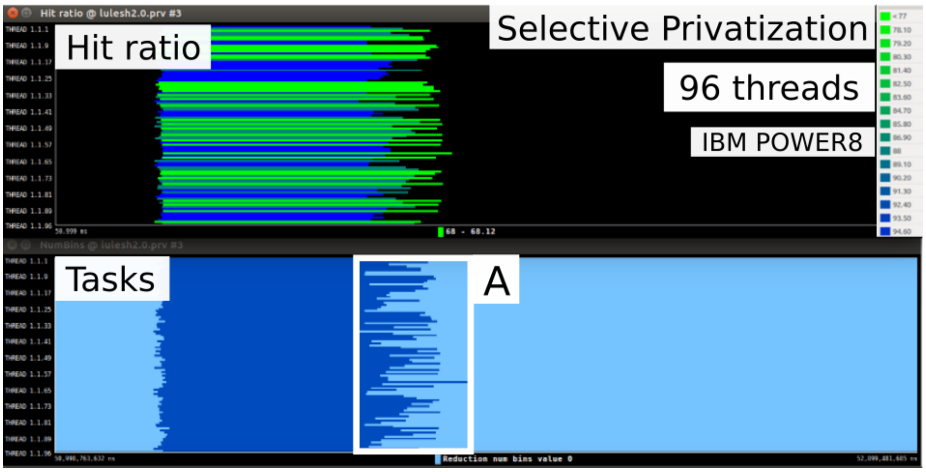
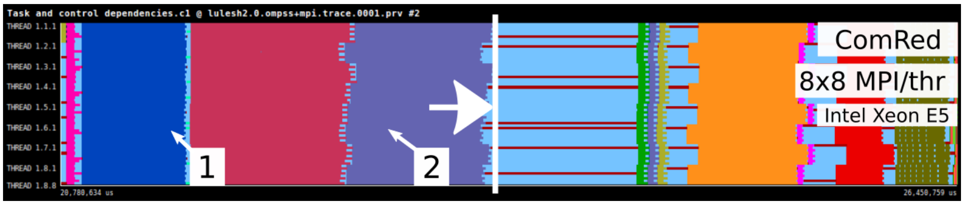
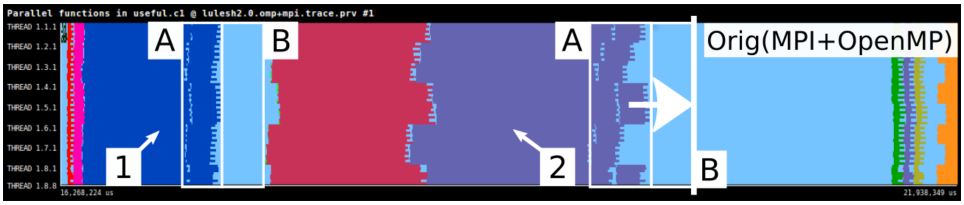


Figure 8.16: *Selective Privatization with PIBOR* results in imbalanced hardware utilization as some tasks can avoid up to 94% of redirections while others cannot.



(a) MPI with tasking in OmpSs



(b) MPI with worksharing in OpenMP

Figure 8.17: Execution traces of both reduction kernels of LULESH on the MareNostrum supercomputer in an 8x8 configuration of MPI processes and threads shows the performance benefit of commutative reductions in OmpSs that avoids the reduction phase (A) and freeing of memory (B).

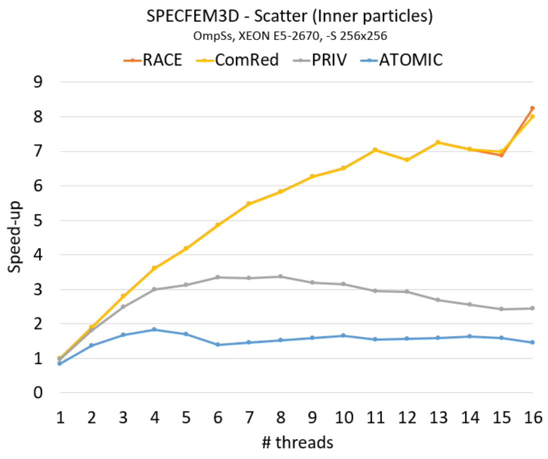


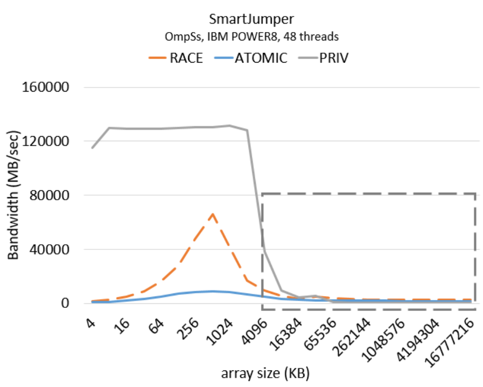
Figure 8.18: Performance scalability of the SPECFEM3D reduction kernel shows speedup relative to the serial execution time with different support techniques including *ComRed*.

shows performance speedups over different thread counts on the Intel XEON E5 processor. Comparably to LULESH, SPECFEM3D performs best with *ComRed*. Performance results do not include the inspection phase.

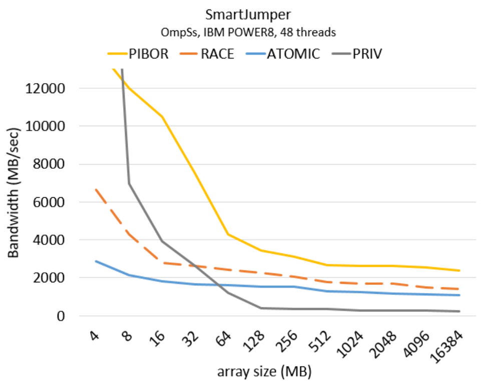
8.4 Performance Evaluation of SmartJumper

SmartJumper implements an irregular reduction over an array type. Since updates are scattered over the entire address range of a potentially large reduction array, we showcase the use of PIBOR as a support technique. PIBOR, Privatization with In-lined Block-ordering, falls within the group of techniques with access redirection and implements redirection into bins. This technique is discussed in Chapter 6 in more detail.

Figure 8.19 shows performance results as achieved memory bandwidth for the SmartJumper micro benchmark. The chart shows results over different array sizes on the IBM POWER8 system. We have selected a thread configuration that reaches the highest memory throughput on the system which is 48 threads. Figure 8.19(a) shows the achieved memory bandwidth for reduction array sizes ranging from 4KB to 16GB. For small data sizes, array replication (*PRIV*)



(a)



(b)

Figure 8.19: Achieved memory bandwidths obtained from different implementations of the SmartJumper reduction kernel show that array replication is a viable technique for small data sizes. *PIBOR* achieves the highest memory bandwidth for array sizes larger than 4 MB.

achieves a satisfactory performance. In this configuration, two threads share one L3 cache of 8MB in size. This corresponds to the steep performance drop when the array size is approaching the 4MB mark and runs out of cache for larger sizes. Figure 8.19(b) shows an augmented view of the previous chart with array sizes ranging from 4MB to 16GB. For these sizes, PIBOR achieves the highest performance due to improved data locality. Further performance statistics that include cache data misses for these benchmark runs are pending.

8.5 Conclusion

In this Chapter we presented performance results for LULESH, SPECFEM3D and SmartJumper. LULESH and SPECFEM3D are representative cases of array-type reductions in scientific computing that exhibit a near-linear access pattern. This access pattern originates from the linear relationship between simulation domains. However, concurrent execution is not trivial due to accesses to overlapping memory regions that arise from parallelization and concurrent execution of the reduction kernels.

We showed that among techniques that implement access redirection, AML techniques perform best (PIBOR) for irregular array-type reductions. This is the case for the SmartJumper micro benchmark, where this technique can avoid excessive allocation of thread-private array copies for larger array sizes.

The use of the original parallel code in LULESH allowed us to make a relevant comparison as that code represents an optimized version developed with knowledge of the mathematical background. This comparison supports our claim that AML techniques are a viable technique in the category of techniques with access redirection.

The highest speedups in our evaluation were obtained with Commutative Reductions (ComRed) for benchmarks with near-linear access patterns. This applies to executions on the Intel Xeon and the IBM POWER8 systems as well as to executions on the MareNostrum cluster. This technique implements an alternative iteration space (AIS) through scheduling. ComRed avoids overheads caused by redirection into private storages, allocation of data copies as well as the use of atomic updates. However, this technique requires an inspector. The use of an inspector has a visible performance impact as shown in execution traces. Still, in LULESH, the cost of inspection can be amortized in approximately 4 iterations of the global simulation loop on our test systems. Given that simulation codes typically run much higher loop counts, we consider that ComRed can substantially improve accumulated application performance.

CHAPTER 9

Conclusion

This work advances research on algorithmic reductions. We have improved their programmability by adding support for task-parallel and array-type reductions.

Task-parallel reductions occur in while-loops and recursive algorithms. While for each recursive algorithm an iterative formulation exists, while-loop programs represent a super class of for-loop computable programs and therefore cannot be transformed or substituted. This limitation requires an explicit support for reduction algorithms that fall within this class. Since tasks are suited for a concurrent formulation of these algorithms, the presented work focuses on language extension to the task construct in OmpSs and OpenMP.

In the first section of this work we have presented a generic support for task-parallel reductions in OmpSs and OpenMP and introduced the ideas of reduction scope, reduction domains and static and on-demand memory allocation.

With this foundation and the feedback received from the OpenMP language review board, we have developed a formalized proposal to add support for task-parallel reductions in OpenMP. This work not only covers the formal specification and evaluation as presented in this document, but also includes substantial effort towards the acceptance of our proposal in form of meetings and draft improvements. This engagement took place in collaboration with my colleagues from the Barcelona Supercomputing Center and led to a fruitful outcome as our proposal has been accepted into OpenMP.

Support for array-type reductions in OmpSs and OpenMP requires software techniques that offer scalability, transparency and extensibility and that are suitable for inclusion into a parallel programming model. With transparency, we refer to the intuitive understanding of behavior by the programmer. Extensibility can be achieved by defining APIs that allow platform vendors to develop optimized solutions easily.

As a first step towards support of array-type reduction in a task-parallel programming model, we present a landscape of support techniques and group them by their underlying strategy. Techniques follow either the strategy of direct access (atomics), redirection or iteration ordering. We call techniques that implement redirection into thread-private data containers as techniques with alternative memory layouts (AMLs) and techniques that are based on iteration ordering as techniques with alternative iteration space (AIS). A universal support of AML-based techniques in parallel programming models can be achieved by defining basic interface methods *allocate*, *get* and *reduce*.

As examples for new techniques that implement this interface, we have developed CachedPrivate and PIBOR. CachedPrivate implements a software cache to reduce communication caused by irregular accesses to remote nodes on distributed memory systems. PIBOR implements Privatization with Inlined Block-ordering, a technique that improves data locality by redirecting accesses into thread-local bins. Both techniques implement a *get-method* that returns a private memory storage for each update operation of the reduction loop.

As an example of a technique with an alternative iteration space (AIS), we present Commutative Reductions (ComRed). This technique uses an inspector-executor execution model to generate knowledge about memory access patterns and memory overlaps between participating tasks. This information is used during the execution phase to schedule tasks with overlaps commutatively. We show that this execution model requires only a small set of additional language constructs.

Performance results obtained throughout different chapters of this work demonstrate that software techniques can improve application performance by roughly a factor of 2-4. Other optimizations in software may include improvements of the runtime implementation such as avoiding polling-based mechanisms used in the implementation of ComRed.

Due to the iterative nature of execution of many scientific applications, we believe that the inspector-executor execution model is future relevant and would allow different optimizations. Its use in the context of reduction techniques or as a foundation to automate decision making between techniques in the future are just two use-case. Further use of this execution model may avoid redundant runtime activities such as dependency computation for certain iterative algorithms.

We believe that hardware support for reductions is desirable. Exploiting their mathematical properties allows weaker coherence requirements between caches and memories which may be used to reduce their latency and bandwidth requirements.

Automated selection of reduction techniques, improved runtime efficiency through the inspector-executor execution model as well as exploring the implications of reductions on hardware architectures are subject to our future work.

Bibliography

[1] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0, July 2013. 2, 31, 45, 48, 50, 89

[2] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254. 4, 104, 121

[3] H. Han, C.W. Tseng, et al. A comparison of parallelization techniques for irregular reductions. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01), San Francisco, CA*, page 27, 2001. 6, 23, 68

[4] Barcelona Supercomputing Center. OmpSs Specification, April, 25th 2014. 12

[5] D. Komatitsch and J. Tromp. Introduction to the spectral-element method for 3-D seismic wave propagation. 139(3):806–822, 1999. 13, 30, 104, 121

[6] J. Ciesko. SmartJumper, https://github.com/janciesko/SmartJumper, May 2016. 13, 124

[7] V. Aggarwal, Y. Sabharwal, R. Garg, and P. Heidelberger. HPCCRandomAccess benchmark for next generation supercomputers. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11, May 2009. 13, 88, 124

[8] GNU.org. Using the GNU Compiler Collection For GCC Version 6.3.0, 2013. 17

[9] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM. 18

[10] Eladio Gutiérrez, Oscar Plata, and Emilio L. Zapata. Data partitioning-based parallel irregular reductions: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(2-3):155–172, January 2004. 23

[11] E. Gutierrez, O. Plata, and E. L. Zapata. Improving parallel irregular reductions using partial array expansion. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 56–56, Nov 2001. 23

[12] H. Yu and L. Rauchwerger. Adaptive reduction parallelization techniques. In *Proceedings of the 14th international conference on Supercomputing*, pages 66–77. ACM, 2000. 23, 68

[13] Hao Yu, Francis Dang, and Lawrence Rauchwerger. *Parallel Reductions: An application of adaptive algorithm selection*, pages 188–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. 23

[14] J.H. Ahn, M. Erez, and W.J. Dally. Scatter-add in data parallel architectures. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 132–142. IEEE, 2005. 24

[15] M.J. Garzarán, M. Prvulovic, Y. Zhang, A. Jula, H. Yu, L. Rauchwerger, and J. Torrellas. Architectural support for parallel reductions in scalable shared-memory multiprocessors. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 243–254. IEEE, 2001. 24

[16] BSC - Parallel Programming Models group. Mercurium C/C++ source-to-source compiler, May 2014. 25

[17] BSC - Parallel Programming Models group. Nanos++ Runtime Library, May 2014. 25

[18] Barcelona Supercomputing Center. Mercurium C/C++ source-to-source compiler, May 2014. 40

[19] Charles E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM. 45

[20] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA'09, pages 79–90, New York, NY, USA, 2009. ACM. 45

[21] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40(10):519–538, October 2005. 46

[22] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: A Unified Deadlock-Free Construct for Collective and Point-to-point Synchronization. In *ICS 08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM. 46

[23] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *Parallel and Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, Rome, Italy, May 2009. IEEE. 46

[24] Jan Ciesko, Sergi Mateo, Xavier Teruel, Vicenç Beltran, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Task-parallel reductions in OpenMP and OmpSs. In *10th International Workshop on OpenMP, IWOMP 2014*, page 1–15, Salvador de Bahia, Brazil, Sep 2014. Springer, Springer. 50

[25] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: An Unbalanced Tree Search Benchmark. In George Almási, Calin Cascaval, and Peng Wu, editors, *LCPC 2006: Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing*, volume 4382 of *LNCS*, pages 235–250. Springer, 2007. 58

[26] A. Duran, E. Ayguadé, R.M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011. 66, 69

[27] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. 66

[28] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009. 66

[29] H. Han and C.W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 393–400. IEEE, 1998. 68

[30] A. Duran, R. Ferrer, M. Klemm, B. de Supinski, and E. Ayguadé. A proposal for user-defined reductions in OpenMP. *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, pages 43–55, 2010. 69

[31] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013. 124