http://www.diva-portal.org

Preprint

This is the submitted version of a paper presented at *10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014.*

Citation for the original published paper:

Podobas, A., Brorsson, M., Vlassov, V. (2014)
TurboBŁYSK: Scheduling for improved data-driven task performance with fast dependency resolution.
In: *Using and Improving OpenMP for Devices, Tasks, and More: 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings* (pp. 45-57).
Springer
Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)
http://dx.doi.org/10.1007/978-3-319-11454-5_4

N.B. When citing this work, cite the original published paper.

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-11454-5_4

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-161691

TurboBŁYSK: Scheduling for Improved Data-driven Task Performance with Fast Dependency Resolution

Artur Podobas[1], Mats Brorsson[1], and Vladimir Vlassov[1]

KTH, Royal Institute of Technology , Stockholm, Sweden
{podobas,matsbror,vladv}@kth.se

Abstract. Data-driven task-parallelism is attracting growing interest and has now also been added to OpenMP (4.0). This paradigm simplifies the writing of parallel applications, extracting parallelism and the use of distributed memory architectures. While the programming model itself is becoming mature, a problem with current run-time scheduler implementations is that they require very large task granularity in order to scale, something which goes at odds with the idea of task-parallel programing where programmers should be able to concentrate on exposing parallelism with little regard to the task granularity. To mitigate this limitation, we have designed and implemented a highly efficient run-time scheduler of tasks with explicit data-dependence annotations: TurboBŁYSK. We propose a novel `pattern-saving` mechanism that allows the scheduler to re-use previously resolved dependency patterns, based on programmer annotations, enabling programs to use even the smallest of tasks to scale well. We experimentally show that our techniques in TurboBŁYSK can achieve nearly twice the peak-performance compared with other run-time schedulers. Our techniques are not OpenMP specific and could be implemented in other task-parallel frameworks.

1 Introduction

The idea behind data-driven task-parallelism is to automatically derive inter-task dependencies based on their use of data-regions. Data-driven task-parallelism extends the fork-join model (for example Cilk [7]) by allowing out-of-order execution of tasks, conceptually similar to instruction-level parallelism in out-of-order processors. There is, however, one significant problem with current state-of-the-art data-driven solutions: the granularity of tasks must be large in order to amortize the cost of automatically solving inter-task dependencies. This might not a problem in High-Performance-Computing applications (where data-sets are very large), managing parallelism from smaller workloads is very hard. Some run-time systems solved the overhead cost by forcing the programmer to specify dependencies manually (see for instance OpenSTREAM [14] and OpenUH [9]) but at the same time reducing user-friendliness.

Some major benefits of task-parallel programming is the promise of *composability*, i.e. a task-parallel program component can be arbitrarily composed with other task-parallel program components, and *decoupling the expression of concurrency from the mapping onto available cores* (threads). The latter is important as it enables the possibility to write programs that automatically can benefit from

new hardware generations with more cores. Both of these encourage programmers to create tasks without considering the available resources or scheduling overhead, something current task schedulers cannot handle well.

We present here techniques that improve the the possibility to use fine-grained tasks in practice. While we show results from an OpenMP run-time system, the techniques are general and can be used in any task-based framework supporting data-dependences among tasks. We also had a hypothesis that many task-parallel programs exhibit *static* data dependency patterns that reoccur iteratively and which could be exploited in the task scheduler. Our *novel* approach allows the programmer to specify which tasks are known to have static iterative dependencies. The tasks still undergo the automatic dynamic dependency resolver for the first iteration and maintain all benefits of dynamic data-driven parallelism, but when the dependencies have been resolved once, they are optimized and re-applied at a smaller cost for subsequent iterations.

Our contributions are:

- TurboBŁYSK: a compiler and highly efficient run-time task scheduler capable of handling OpenMP 4.0 tasks,
- an evaluation of TurboBŁYSK against state-of-the-art frameworks supporting data-driven task parallelism,
- a proposed extension of OpenMP 4.0 tasks to support `dep_pattern`s that enable the run-time system to conserve task dependencies between iterations, and
- a detailed overview of the design decisions in *TurboBŁYSK* and why it performs better than similar state-of-the-art implementations for fine-grained parallelism.

On the same hardware platform we nearly double the performance for fine-grained task-parallel programs compared to current state-of-the-art alternative implementations. We will now further motivate the need for this work before describing the implementation in more detail.

2 Motivation

It is computationally expensive to resolve data dependencies dynamically in run-time. In order to quantify this we modified gcc's OpenMP 4.0 run-time system (libgomp in gcc 4.9 snapshot 20131103) to measure the time spent handling dependencies and compared it with the time consumed in other parts of the execution. We created a benchmark that performs Gauss-Seidel computations over a 2048x2048 matrix and executed it with two block-sizes: 128x128 block size for coarse-grained tasks with 6k dependencies to solve, and 24x24 for fine-grained task execution with 180k dependencies to solve.

The results were striking; the time consumed managing dependencies in the coarse-grained case was only 1.2% of the total execution time while the time consumed for dependency resolution in the fine-grained case exceeded 40%. The execution time of the fine-grained execution was even slower than the serial version.

Practically this means that there is no benefit of using multiple cores for executing a Gauss-Seidel on a 256x256 matrix. Other applications behaved in a similar fashion. Our opinion is that programmers should not need to concern

themselves with decomposition granularity of tasks. The task scheduler should work with as fine granularities as possible. Some applications really *require* tasks to be fine-grained in order to exploit parallelism at all. While we realize that it is unrealistic to completely ignore task granularity, we strive to do as best as possible.

In the next section we explain what we do to significantly improve on current state-of-the-art.

3 TurboBŁYSK: A framework for fast-dependency resolution

This section describes the *TurboBŁYSK* framework which implements OpenMP 4.0 and consists of a transcompiler (BłyskCC) and a run-time system (TurboBŁYSK). This includes support for the `depend`-clause which allow for data-driven task execution of disjoint data-regions. *BłyskCC* is a source-to-source compiler which transforms the OpenMP code to run-time system API calls; a hand-written recursive-descend parser converts the source code into an Abstract-Syntax-Tree (AST) where the OpenMP pragmas are transformed and tasks outlined (similar to Mercurium [2]).

3.1 Automatic Dependency Resolution

Dependencies are extracted by keeping track of each task that will work on a specific data-region(s). A data-region is defined by a virtual address coming from the application such as a variable or parts of an array, and the size of that region. Each known data-region is inserted into a *dependency-structure* that keeps track of all tasks that use this particular data-region. There are private dependency-structures for each parent task, allowing hierarchical decomposition of dependent tasks. When a new task is submitted from the application to the run-time system, each data-region the to be used is looked up, obtaining the dependency-structure for that data-region. If the obtained dependency-structure is empty, then the task claims that data-region. If the task only *read*s from that data-region and existing tasks using the region *read*s from it then the dependency is marked as resolved (Read-After-Read). Only Flow- (Read-After-Write), Anti- (Write-After-Read) and Output- (Write-After-Write) dependencies are considered. Note that even if a Read-after-Read dependency is encountered, the task will still need to be entered into the dependency-structure for subsequent tasks.

Fig. 1 shows actions performed by our run-time system when a task is created and we will refer to this figure in the following text.

Task creation When the application creates a task (Fig. 1:a) a run-time system call is performed to submit the task. At this point, if the run-time system finds that the task does not use the `depend` clause it will immediately schedule the task for execution (Fig. 1:g); otherwise, it will proceed to analyze the dependencies (Fig. 1:b).

The run-time system needs to locate the *dependency-structure* that is associated with the task's data-regions. This search (Fig. 1:c) is the most computationally intensive part in resolving dependencies. Multiple threads can be using the dependency-structures, hence they need to be protected by a lock. In many

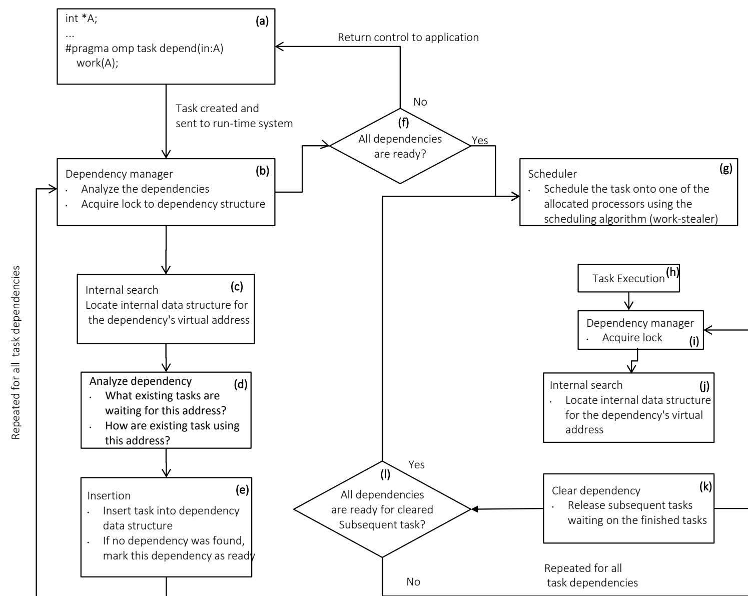


Fig. 1: Flow-chart representing the steps required to solve a task's dependencies including the steps performed when a task finishes executing

run-time systems (for example GCC's libgomp) this lock is global, preventing concurrent access to the dependency-structure. In TurboBŁYSK, we allow partial concurrent access. We keep each dependency-structure sorted in a *bitwise trie* – a binary tree where searching for a dependency-structure is done by traversing all bits in the data-region's virtual address. To allow concurrent access, the root-node for our trie contains 64 independent entries; each of the 64 entries is protected by a lock. The lock is internally represented as a 64-bit variable where each bit represents a lock for one of the 64 entries. Mapping from a data-region to the required lock is done by examining the six least-significant bits of the data-region's *word address*; we found that when task-granularities are fine, tasks tend to work on data-regions whose virtual addresses are close to each other and will therefore use different locks (different value in the least-significant six bits). When a lock has been calculated for each data-region a task will use, the locks are acquired and search in the trie begins – an $O(log(n))$ operation that results in either a found data-region's dependency-structure or the creation of a new dependency-structure. The lock and the found dependency-structure are saved in the task-structure for later use.

Inside the dependency-structure When the dependency structure (Fig. 1:d,e) has been found for a particular data-region, the run-time system will insert the task into the end of a list. The list contains all previous tasks waiting for/or currently using the data-region. The structure contains a variable we call the dependency *q_position*(queuing_position). The *q_position* is increased when we detect a Flow-, Anti- or Output- dependence. The type of dependency is

determined by comparing the current task with the previously inserted tasks (for that data-region). We also maintain a state variable called *a_position* (active position) which corresponds to the position that is currently executing. When all tasks for a given *a_position* finish executing, the run-time system releases subsequently dependent tasks. If a new task enters a dependency region, and the active_position is the same as the queuing_position (*a_position*==*q_position*) and both the tail-task and the new task will *Read* from the data region, then it has no dependencies on other previous tasks using the data-region.

When the dependencies-analysis is complete When a task's data-regions have had their dependencies detected, the run-time system checks if any dependencies were found (Fig. 1:f); if none were found then the task is scheduled for execution (Fig. 1:g). Otherwise, the run-time system returns control to the application.

When task finishes executing When a processor finishes executing a task (Fig. 1:h), the previously saved lock will be re-acquired in order to lock-on to the dependency-structures associated with the finished task (Fig. 1:i,j). We go through each of the recently completed task's dependency structures and reduce the number of tasks in the *active position* (Fig. 1:k). If the tasks in the active position reaches zero, the processor will increase the active position and remove the dependency of subsequent tasks in the list until it detects anything other than a Read-After-Read dependency. If, when clearing dependencies of tasks, a task has no other dependencies in other data-regions then it is scheduled for execution (Fig. 1:l).

3.2 Dependency Pattern

Data-driven parallelism allows the programmer to be oblivious towards dependencies between tasks as they are transparently derived by the run-time system. We can relax this model by allowing the programmer to specify whether a task's sub-graph is static or not. Should a task's sub-graph be static, the run-time system can dynamically extract task dependencies from the first run of the task and re-use them in later invocations. Note that the programmer still does not need to know the dependencies between tasks – she only specifies whether the tasks will have static dependencies.

We propose a new OpenMP keyword called `dep_pattern`, which together with a unique tag enables the run-time system to extract task-graph dependencies for that tag and re-apply them on the next invocation of that task. Consider the following code:

```
void matmul(float *A,float *B, float *C,int DIM) {
    for (int i=0;i<DIM;i+=4)
        for (int j=0;j<DIM;j+=4)
            for (int k=0;k<DIM;k+=4)
#pragma omp task depend(in:A[k+(j*DIM)],B[k+(j*DIM)]) \
                            depend(inout: C[i+(j*DIM)]
                matmul_block( &A[k+(j*DIM],...);
}
```

```
...

#pragma omp task dep_pattern("matmul_16x16") depend(in:A,B) depend
      (inout:C)
      matmul(A,B,C,16);
```

The above example code is a standard blocked matrix multiplication. We create a top-level task to perform the matrix multiplication and specify the data-regions that will be used; in addition, we tell the compiler that this task will contain tasks whose dependencies are static regardless of which virtual addresses they use. When the run-time system encounters a `dep_pattern` clause it will first try to determine if the dependencies are known for the tag from previous invocations. If the run-time system fails to locate the tag ("matmul_16x16" in the example above), it will use the automatic dependency manager to derive the (static) dependencies and sample those. It is up to the programmer to chose the tag's name and use it consistently with the `dep_pattern`-clause– a tag for a wavefront pattern will not work on a e.g. matrix-multiplication and will yield data-races.

Sampling is performed by renaming dependency structures which we refer to as *reservation stations* to further the analogy with out-of-order execution of instructions. Recall that *TurboBLYSK* uses a *bitwise trie* to keep track of all dependency structures. The difference when sampling is that each of the dependency structures is given a unique integer ID and that the run-time system saves what dependency IDs each data-region in each sequentially spawned task used. This information is saved in a stream where each task is represented as: [int]#ID [int]#deps [int]dep1 [int]modifier1. The modifier specifies how the task will access the reservation station. When the task has finished, the raw stream will be processed (optimized). Information in the stream will be losslessly encoded, each integer can potentially be reduced to a nibble in size. Similarly, the modifiers for the reservation stations are encoded in the least significant bits of the static *reservation station* used with the dependency. To further increase the compression and improve performance, the run-time system will perform the following when compressing the stream:

– It will simulate all tasks' dependencies. If a reservation station (data-region) will only be *read* from, it will be completely removed. As an example, a matrix multiplication will always read from two matrices and produce results into a third. The dependencies on the two first matrices will thus be removed when analyzing the stream which results in a 66% reduction in amount of dependencies and reservation stations. It can be argued that those two dependencies can be removed from the source code; doing so would however remove information required to use distributed memory architectures or improved scheduling [11].
– For each task, and its reservation station usage, a lock will be mapped. We look at the histogram of reservation stations that are used the most, and construct the locks such that the most intensively used reservation stations can be worked on concurrently. This results in a better lock mapping compared with the heuristic used in the automatic dependency phase.

An example encoding can be seen in Fig. 2. Here, since the reservation stations used by each dependency fits into a nibble (4-bit), the stream will start with `b1100` indicating three dependencies to follow (`b11`=3) and that the dependencies will be represented in a nibble form specified by the `b00` prefix- other prefixes

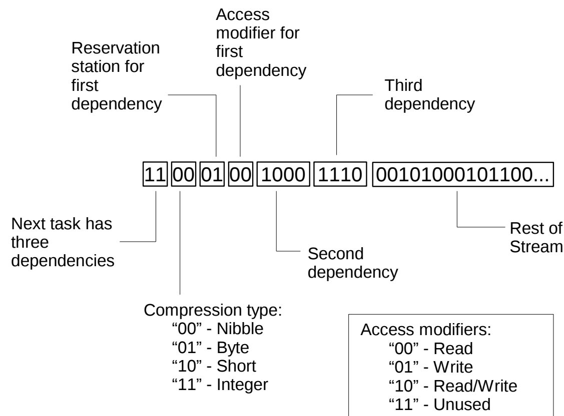


Fig. 2: Example showing a task, its dependencies and the modifiers encoded in the bit stream.

indicate other compressions schemes. What follows are 12 bits (3 x 4-bit) that represent each dependency of the task; the least-significant two bits express the type of dependency (read/write/read-write) and the two most-significant bits represents the *reservation station* to be used.

Using the stream The stream is now encoded and ready to be used. On the next invocation of a task with a "matmul_16x16" tag, the encoded stream will be used to apply the dependencies. The run-time system, knowing that a tag-stream pair exist, will proceed to create a set of reservation stations rather than a trie. When tasks start spawning, the stream is parsed and the dependency-structure is quickly ($O(1)$ instead $O(logN)$) found in a reservation station – unlike the automatic dependency resolver, the locks and addresses need not be analyzed and the cost of searching for a dependency structure in the bitwise trie is completely removed.

Note that using the `dep_pattern` keyword does not remove the ability to know the different data-regions (address ranges) a task uses, nor does it remove the ability to offload tasks onto distributed devices (such as GPUs). Only the dependencies are static. Our approach also allows for other static analyzing techniques to be used on task-graphs, helping the scheduler's decision making. We also allow mixing the `dep_pattern` keyword with tasks' whose dependencies are not static in a hierarchical manner. Finally, the streams could also be saved to persistent storage for use in future executions.

4 Methodology

System Architecture We used a shared memory NUMA multiprocessor with four AMD Opteron 6172 processors totalling 48 cores and 64 GB DRAM. All 48 cores were used in our evaluations. The Operating System used is Red Hat version 6.5 (kernel version 2.6.32).

Comparison We compared our implementation against OmpSs [6] using Mercurium 1.3.5.8 (OmpSs's compiler) and Nanos++ (OmpSs's run-time system)

version 0.7a. We also compared against gcc's OpenMP implementation libgomp, taken from the gcc 4.9 snapshot archive version 3.455. OmpSs is a well-known task-library that, arguably, is one of the pioneers of data-driven computing in the task-based paradigm, which is why we chose to compare against it.

Benchmarks We used five different benchmarks in our performance evaluation. We executed each benchmark/input pair ten times, using the median to eliminate extreme corner cases. Resilience to fine-grained parallelism was evaluated by varying the amount of concurrency exposed, typically by altering the block-size each task works on. The benchmarks are:

- **Matrix Multiplication** Parallel blocked Matrix Multiplication where each task handles a blocked region. We used the ATLAS BLAS [4] library for the multiplication.
- **SparseLU** Parallel LU matrix factorization. Original code from BOTS [5] which we converted to OpenMP v4 and OmpSs.
- **Gauss-Seidel** Parallel, blocked and iterative Gauss-Seidel calculation derived from previous studies [14, 13].
- **Cholesky** factorization based on the ATLAS BLAS library [4].
- **N-Body** a parallel version of this classical problem.

Schedulers To ensure fairness, both TurboBŁYSK and OmpSs used the same work-stealing baseline scheduler. Each processor has its own queue to which is tasks are spawned. Processors unable to find work in their own queue will attempt to steal work from other processors. Victim selection is pseudo-random. The libgomp scheduler uses a global task queue.

Speedup Calculations We calculated the speedup when running our evaluated run-time systems by normalizing it against the *fastest* serial version we found: $Speedup = {}^{t_{ser}} / {}_{t_{par}}$ where t_{ser} is the time taken for the serial version, and t_{par} is the parallel makespan (time taken for the parallel region). Initialization phases (e.g. memory allocation or `dep_pattern`[1] stream encoding) are not included in the evaluation.

5 Results

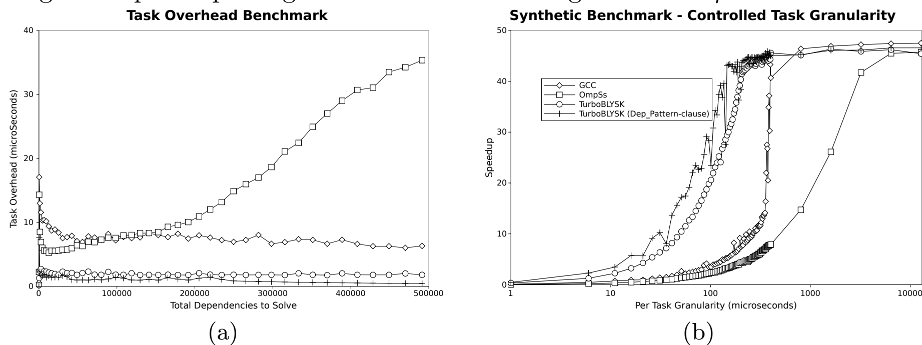
5.1 Resilience to Fine-Grained parallelism

We constructed a micro-benchmark where we simulate a wave-front pattern of dependencies. Each task has a read dependency on the north and west elements in a matrix. The benchmark iterates five times over the matrix. We first evaluated the task-overheads (Fig. 3:a). This was done by creating empty tasks and thus isolating the run-time system's management of the tasks and their dependendencies. As we gradually increased the wave-front matrix (increasing the number of dependencies), we found that TurboBŁYSK has a 1.88μs task-overhead compared to gcc's 7.38μs and OmpSs's 19.26μs. The `dep_pattern` clause had an

[1] Which is fair, since the stream is encoded once and can be used forever.

even smaller average overhead of 1.21μs, where 0.42μs task-overheads could also be observed. Also note that TurboBŁYSK, remains fairly constant independent (1.88$\pm$0.21μs) of how many dependencies the application has.

We also simulated an artificial work-load in each task to make it possible to vary task granularity while keeping the same number of tasks. The purpose is to see at what granularity each framework starts to scale efficiently. We fixed the micro-benchmark to have 245,760 dependencies to solve. We found (Fig. 3:b) that TurboBŁYSK started to scale already with 6μ long tasks. Both the automatic-dependency solver and the `dep_pattern`-clause outperformed all other implementations, reaching peak-performance much earlier than the other implementations. Gnu C's libgomp starts scaling when the task granularities reach 35μs, but the performance is not very stable until granularities of 42μs or longer. OmpSs requires granularities that are larger than 300μs to scale well.



(a) (b)

Fig. 3: (a): Average task-overhead with different implementations. Lower is better. (b): Speedup with different controlled granularities on tasks. Higher is better.

5.2 Experimental results

Fig. 4 shows speedup profiles for all the evaluated benchmarks. Overall, our run-time system performs consistently better than both OmpSs and gcc's libgomp: TurboBŁYSK starts to scale at much lower granularities and to a higher number of cores in most cases; the reason for this increase in scalability is that there simply is not enough parallelism exposed at the granularities required by the other run-time systems.

For example, SparseLU (Fig. 4:c) using the `dep_pattern` clause allows scaling already with a block-size of 18x18 while gcc's libgomp and OmpSs require a block-size of 30x30 and 25x25 respectively – a similar behaviour is seen for all the benchmarks. The results for Cholesky (Fig. 4:d) shows that the peak-performance of our run-time system executes nearly twice as fast as for the other run-time systems, and four times as fast using the `dep_pattern`-clause.

Many of the evaluations showed a trend where the automatic dependency-manager (BlyskMP) and the `dep_pattern`-clause (TurboBŁYSK (Dep_Pattern)) had equal performance, diverging only when the granularity of tasks was so small that even our fast automatic dependency-manager failed to handle them. Using the `dep_pattern` clause, performance could be maintained or even increased at these fine granularities (Fig. 4:a,b,c,e).

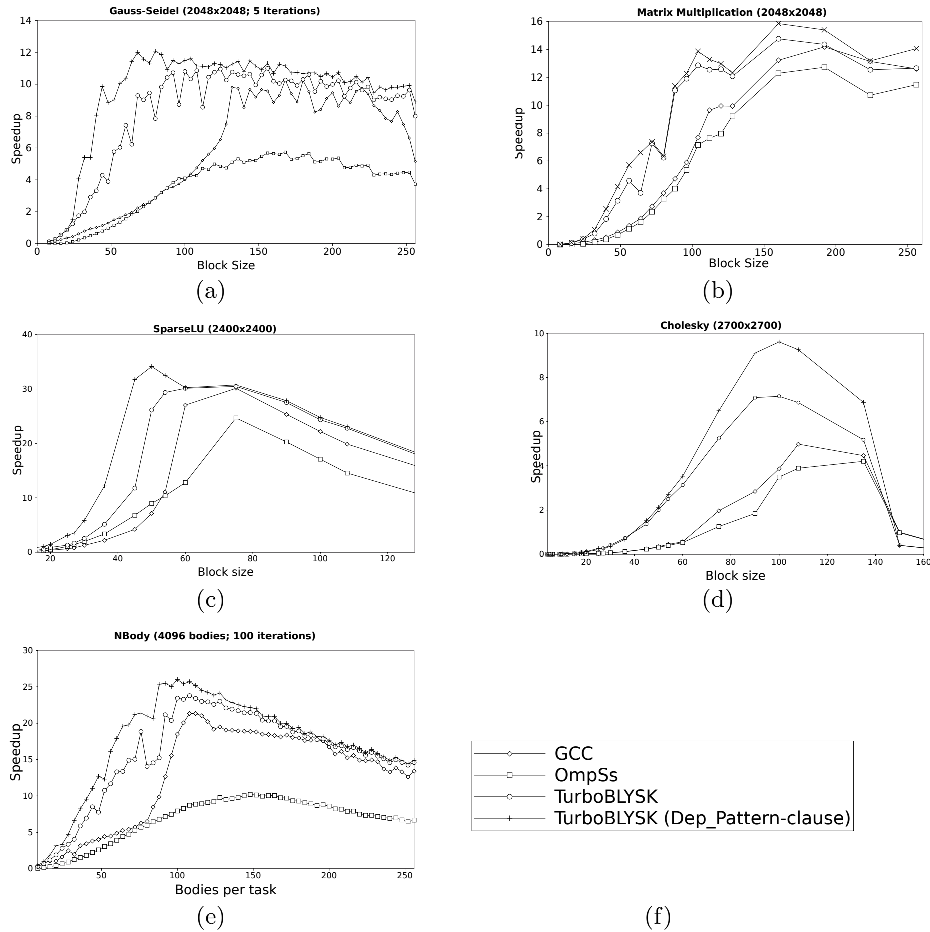


Fig. 4: Experimental evaluation of TurboBŁYSK, gcc's OpenMP 4.0 implementation and OmpSs. We increase the task granularity (reducing parallelism) moving from left to right on the x-axis. Speedup–how much faster parallel execution is over the serial– is seen on the y-axis

6 Related Work

Vandierendonck et. al [16] compared different methods of dependency-analysis. Our implementation of the dependency-management is very similar to their Hypergraph scheme, with minor implementation differences. The evaluation took place in SWAN and their prototype run-time system based on data-driven parallelism but which does not focus on automatic dependency management on virtual addressesn (which we do). Our `dep_pattern` clause receive the same level of algorithm complexity as their algorithm, while still conveying information to the run-time system regarding virtual addresses.

The OSCAR compiler [12] supports a similar concept to our, but in the fork-join paradigm. *During compilation*, static tasks are identified and statically scheduled to optimize cache behavior. However, our work has two differences: we use dynamic dependencies to figure out static properties of a Direct-Acyclic-Graph and our method uses a different paradigm (data-driven) where compiler detected strategies are significantly harder (if possible).

OmpSs [6] is an OpenMP based run-time system and compiler framework targeting data-driven parallelism. OmpSs is a merge between a data-driven model, StarSs [10], and OpenMP. OmpSs supports both region based [13] and address-based (as in the present study) data-driven parallelism, and also has GPU-support.

OpenUH [9] supports task dependencies using IDs [9] that are used to specify dependencies between tasks and removes the overhead of dynamically finding the dependencies, but expects more from the programmer. Conceptually our `dep_pattern` clause converts a dynamic data-driven task graph to a graph such as OpenUHs.

X-Kaapi [8] is a OpenMP-like parallel model supporting multiple paradigms, amongst others data driven parallelism. Similar to OmpSs and the OpenMPv4 specification, the user conveys information regarding the use of memory regions to the run-time system, which inserts dependencies between different tasks. X-Kaapi also supports the removal of WAR-dependencies by renaming regions. X-Kaapi is used in the libKOMP [3] run-time system, featuring OpenMP like execution.

OpenSTREAM [14] is a run-time system featuring *streams* in which data flows between different tasks. It uses the streams to detect producer-consumer patterns between tasks (dependencies). OpenSTREAM was evaluated against StarSs, showing what a large impact the dependency management had on the execution time of fine-grained parallel application.

StarPU [1] is a framework for both accelerators and homogeneous general-purpose systems. It supports the notion of data-regions which the run-time system exploits to provide efficient multi-GPU performance. It uses a variant of the HEFT [15] to schedule workloads on resources using a model based on the size of the input regions.

7 Conclusions

We presented TurboBŁYSK: an OpenMP4.0 tasking framework for data-driven parallelism, which we used to show that even the finest of granularities can scale well. We also introduced the `dep_pattern`-clause; a novel and intuitive method of re-using dependency patterns commonly found in task-based application. Using the `dep_pattern`-clause programmers can further leverage the performance. Our experimental evaluation verified all our methods, which we compared to existing state-of-the-art models.

Acknowledgements

This work was funded by the Artemis PaPP Project nr. 295440. We thank the researchers Alejandro Rico and Alex Ramirez from Barcelona Supercomputer Center for sample OmpSs applications used in this study. Further thanks goes to Xavier Teruel for answering questions related to Nanos++/OmpSs regarding dependencies. Thank you!

References

1. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.
2. J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos Mercurium: a research compiler for OpenMP. In *Proceedings of the European Workshop on OpenMP*, volume 8, 2004.
3. F. Broquedis, T. Gautier, and V. Danjean. Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. In *OpenMP in a Heterogeneous World*, pages 102–115. Springer, 2012.
4. R. Clint Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3–35, 2001.
5. A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 124–131. IEEE, 2009.
6. Duran, Alejandro and Ayguadé, Eduard and Badia, Rosa M and Labarta, Jesús and Martinell, Luis and Martorell, Xavier and Planas, Judit. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.
7. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.
8. T. Gautier, F. Lementec, V. Faucher, and B. Raffin. X-Kaapi: a Multi Paradigm Runtime for Multicore Architectures. Rapport de recherche RR-8058, INRIA, Feb. 2012.
9. P. Ghosh, Y. Yan, and B. Chapman. Support for dependency driven executions among openmp tasks.
10. Labarta, Jesus. StarSS: A programming model for the multicore era. In *PRACE WorkshopNew Languages & Future Technology Prototypes at the Leibniz Supercomputing Centre in Garching (Germany)*, 2010.
11. Muddukrishna, Ananya and Jonsson, Peter A and Vlassov, Vladimir and Brorsson, Mats. Locality-Aware Task Scheduling and Data Distribution on NUMA Systems. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 156–170. Springer, 2013.
12. H. Nakano, K. Ishizaka, M. Obata, K. Kimura, and H. Kasahara. Static coarse grain task scheduling with cache optimization using openmp. In *High Performance Computing*, pages 479–489. Springer, 2006.
13. Planas, Judit and Badia, Rosa M and Ayguadé, Eduard and Labarta, Jesus. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.
14. Pop, Antoniu and Cohen, Albert. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):53, 2013.
15. H. Topcuoglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.
16. H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. Analysis of dependence tracking algorithms for task dataflow execution. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):61, 2013.