A Context-Aware Primitive for Nested Recursive Parallelism

Herbert Jordan[1](✉), Peter Thoman[1], Peter Zangerl[1], Thomas Heller[2], and Thomas Fahringer[1]

[1] University of Innsbruck, Innsbruck, Austria
{herbert,petert,peterz,tf}@dps.uibk.ac.at
[2] Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
thomas.heller@fau.de

Abstract. Nested recursive parallel applications constitute an important super-class of conventional, flat parallel codes. For this class, parallel libraries utilizing the concept of tasks have been widely adapted. However, the provided abstract task creation and synchronization interfaces force corresponding implementations to focus their attention to individual task creation and synchronization points – unaware of their relation to each other – thereby losing optimization potential.

Within this paper, we present a novel interface for task level parallelism, enabling implementations to grasp and manipulate the context of task creation and synchronization points – in particular for nested recursive parallelism. Furthermore, as a concrete application, we demonstrate the interface's capability to reduce parallel overhead within applications based on a reference implementation utilizing C++14 template meta programming techniques to synthesize multiple versions of a parallel task during the compilation process.

To demonstrate its effectiveness, we evaluate the impact of our approach on the performance of a series of eight task parallel benchmarks. For those, our approach achieves substantial speed-ups over state of the art solutions, in particular for use cases exhibiting fine grained tasks.

1 Introduction

For the development of parallel programs, various programming language extensions and libraries have been created. Many of these, including OpenMP, MPI, OpenCL, or CUDA, focus on the concept of parallel loops, and variations of those, as their primary use case. In general the associated data parallelism provides high degrees of concurrency, leading to scalable applications. Furthermore, the management overhead for distributing sub-ranges of parallel loops scales only with the number of processors, not the problem size itself – and is thus low.

© Springer International Publishing AG 2017
F. Desprez et al. (Eds.): Euro-Par 2016 Workshops, LNCS 10104, pp. 149–161, 2017.
DOI: 10.1007/978-3-319-58943-5_12

However, beyond the class of loop-parallel applications – also referred to as *flat* parallel applications – there is a large group of algorithms favoring nested parallelism. In those, concurrent control flows spawn further, *nested*, parallel control flows to obtain higher degrees of parallelism. In many cases, this nesting of parallel control flows, or *threads*, is even continued recursively.

Example algorithms benefiting from nested parallelism include divide and conquer approaches such as those found in sorting algorithms, the entire class of graph processing, as well as the wide range of problem space exploration algorithms, covering combinatorial problems, optimization problems, and decision problems (e.g. SAT or SMT problems). Also, many numerical problems have effective nested parallel implementations: matrix multiplication – in its 3-loop form a text book example for loop-level parallelism – can be more effectively solved by Strassen's algorithm, which exhibits a nested recursive parallel structure. Furthermore, conventional flat parallelism constitutes a special case of nested parallelism. Thus, the class of flat parallel algorithms is a true subset of the class of nested parallel algorithms.

Due to its benefits, existing languages and libraries have been modified to provide support for nested parallel codes. OpenMP introduced task-based parallelism in its version 3.0 [3] and CUDA supports nested parallelism since version 5.0 [8]. However, both superimpose nested parallelism on their otherwise flat execution model, resulting in management overhead for the runtime as well as for the developer. Cilk, on the other hand, has been specifically designed for nested, recursive parallelism, resulting in a (nearly) hands-off solution for the scheduling of nested (recursive) parallelism. However, as we will address in this paper, Cilk's fully general approach introduces overhead. Furthermore, as a compiler based approach, modifications and extensions to Cilk require modifications in the compiler and are thus not portable among different system software stacks.

The construct presented in this paper, the *prec* operator, provides a way to define nested parallel operations offering the flexibility of future based task systems, combined with the hands-off scheduling and load management qualities of Cilk, yet avoiding Cilk's inherent overhead for task-scheduling opportunities. Furthermore, all of those features are realized utilizing C++'s template-metaprogramming feature – essentially a built-in, widely supported language feature to script C/C++ code generation in an early compilation stage. Thus, while being a code generation based solution, its implementation behaves like a library. It can therefore be flexibly extended or modified and is directly supported by every standard C++ compiler. Consequently, parallel codes developed utilizing *prec* are portable to all systems offering a C++ compiler.

2 Motivation and Main Idea

Our work was motivated by the unexpected low parallel performance observed when parallelizing nested recursive algorithms using state-of-the-art tools. For instance, Fig. 1 compares the execution time of various parallel codes computing Fibonacci numbers recursively similar to

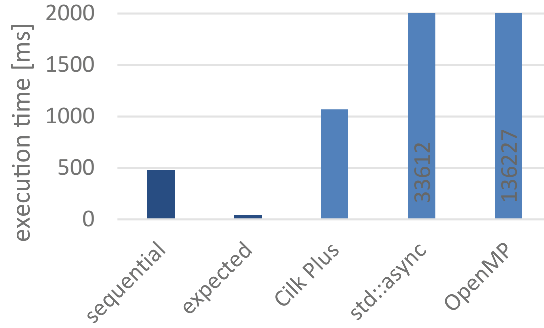

Fig. 1. Performance comparison of parallel `fib(40)` computations using parallel libraries integrated into GCC 5.3.0, compared to a desired linear speedup.

```
int fib(int n) {
    if (n <= 2) return 1;
    int a = spawn fib(n-1);
    int b = fib(n-2);
    sync;
    return a + b;
}
```

utilizing different parallel library and runtime implementations. Based on the parallel structure of the problem, an almost linear speedup would have been expected. However, as the data shows, none of the parallel implementations manages to provide any speedup over the sequential version. While the Cilk based version exhibits a slowdown by a factor of 2.2, the `std::async` and OpenMP variants lead to a slowdown by a factor of 70 and 284 respectively.

One common concept in all existing parallel libraries or program extensions is that each creation of at task is handled independently. While it is very prominent and explicit when using C++11's `std::async`, Cilk and OpenMP implementations also process each individual task-creation site independently of its context. Thus, if nested in a recursive function, each (potential) task-creation point has to be processed, and the cost for the thereby introduced overhead has to be paid.

With the `prec` operator, to be specified in detail in Sect. 3, we enable the parallel library implementation to grasp the context of a parallel task creation point – in particular in a nested recursive environment. Furthermore, we enable the parallel library to specialize the context around those task creation points. This capability is utilized by the `prec` operator to create multiple implementation versions of a given recursive operation – one processing sub-tasks in parallel and another processing sub-tasks sequentially. The runtime system may then switch between these two implementations depending on the system state.

As a result, when utilizing the `prec` operator for parallelizing nested recursive code, *the compiler is synthesizing an interleaved sequential and parallel version of the recursive function from a single specification.* Thus, the user only has to provide and maintain a single implementation. The multi-versioning is conducted

by the compiler. Furthermore, the compiler is capable of applying low-level optimizations to the sequential version of the synthesized code such that, compared to a purely sequential version, no performance is lost.

3 Method

Let α, β, γ, and δ be type variables, $A \rightarrow B$ denote the type of a function accepting a value of type A as an argument and obtaining a value of type B, and $(A_1, \ldots, A_n)$ be the type of a n-tuple where the i-th component is of type A_i. Further, let $\mathcal{B}$ be the type of the boolean values *true* and *false*, and $\mathcal{N}$ the set of natural numbers.

3.1 The rec Operator

A recursive function can be defined by providing (i) a test for the base-case, (ii) a function evaluating a base-case, and (iii) a function evaluating a step-case. Furthermore, in a typed system, the parameter type and the result type of the function has to be specified. For instance, for defining a recursive version of the Fibonacci function fib, where

$$\text{fib}(x) = \begin{cases} 1 & x <= 2 \\ \text{fib}(x-1) + \text{fib}(x-2) & \text{otherwise} \end{cases}$$

the parameter and result type is $\mathcal{N}$ (natural numbers), the base-case test, the base-case, and the step case are given (in C++11 lambda-like syntax) by the three expressions

```
base_case_test  = [](𝒩 x){ return x <= 2; };
base_case       = [](𝒩 x){ return 1; };
step_case       = [](𝒩 x, 𝒩→𝒩 fib){
                        return fib(x-1) + fib(x-2);
                  };
```

where the parameter `fib` is a token passed as an argument to conduct recursive calls.

Let α and β be type variables representing the parameter type and result type respectively. Then, in the general case, a definition requires

- a base-case test of type $\alpha \rightarrow \mathcal{B}$
- a base-case evaluation function of type $\alpha \rightarrow \beta$ and
- a step-case evaluation function of type $(\alpha, \alpha \rightarrow \beta) \rightarrow \beta$

To combine those parameters, a higher-order function `rec` of type

$$(\alpha \rightarrow \mathcal{B}, \alpha \rightarrow \beta, (\alpha, \alpha \rightarrow \beta) \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$$

can be defined, such that a call `rec(a,b,c)` evaluates to the recursive function

```
β f ( α in ) {
    if (a(in)) return b(in);
    return c(in,f);
}
```

Thus, for our example above, we obtain after inlining

```
𝒩 f ( 𝒩 in ) {
    if (in <= 2) return 1;
    return f(in-1) + f(n-2);
}
```

corresponding to a sequential implementation of the Fibonacci function.

3.2 The prec Operator

For the parallel case it would be desirable to process sub-tasks of the recursion in parallel. However, once sufficient concurrent control flows are present in the system, the individual tasks should avoid the overhead induced by allowing for task spawning by being processed sequentially. Thus, the scheme of the `rec` operator needs to be modified to enable the generation of sequential and parallel versions of the recursive function as well as to enable the task scheduler to decide which version to execute at every scheduling point.

Similar to the `rec` operator, the `prec` operator is a higher order function of type

$$(\alpha \to \mathcal{B}, \alpha \to \beta, (\alpha, \alpha \to \gamma\langle\beta\rangle) \to \beta) \to (\alpha \to \text{future}\langle\beta\rangle)$$

accepting three functions as arguments, and returning a new function as a result. The interpretation of α and β is the same as for the `rec` operator. Additionally, γ is a generic type to be substituted by a value wrapper providing access to a (potentially asynchronously processed) value. Two examples of such value wrappers are:

- `value<`δ`>` wrapping an immediately available value of type δ that has been computed synchronously, and
- `future<`δ`>` referencing a value of type δ asynchronously computed by a task

The result of the `prec` operator is a function asynchronously computing the recursive function defined by its parameters, thus returning a future to a value of type β.

Let $\text{expr}_1, \ldots, \text{expr}_n$ be $n \geq 1$ expressions of type T. Furthermore, let the expression `spawn expr`$_1$`or ... or expr`$_n$ create a task evaluating asynchronously exactly one of the given n expressions and return a future of type `future`$\langle T\rangle$ as a handle to the resulting value. A call to $\text{prec}(a, b, c)$ is translated into the nested recursive parallel function `f` defined by

```
value<β> seq_f( α in ) {
  if (a(in)) return value(b(in));
  return value(c(in,seq_f));
}
```

```
future<β> par_f( α in ) {
  if (a(in)) return spawn b(in);
  return spawn c(in,f);
}
future<β> f( α in ) {
  return spawn seq_f(in).get() or par_f(in).get();
}
```

where `seq_f` is the sequential version of the recursion, `par_f` the parallel version and `f` a version serving as a dispatcher point between the sequential and parallel version upon each recursive invocation. Note that the functions `par_f` and `f` are mutually recursive, while `seq_f` is only invoking itself.

3.3 The Runtime System

A runtime system supporting the `prec` operator needs to provide efficient implementations for the `spawn` expression and the `future` class. It can rely on the fact that the `spawn` expression is called by passing (i) a single, nested parallel expression or (ii) two expressions, where the first is a sequential implementation and the second a parallel implementation. Thus, in case two implementations are provided, the first may be used in situations where the available parallel processing units are saturated and more parallelism is not beneficial, while the second implementation may be chosen if there are still idle resources in the system.

For the futures, any runtime implementation has to provide means to synchronize upon the completion of spawned tasks as well as to retrieve asynchronously computed values.

3.4 Implementation

We have implemented the `prec` operator in a reference implementation utilizing C++14 and its template meta-programming facilities. It is internally maintaining a pool of threads, each equiped with a local task queue. For load balancing, a task steeling policy has been integrated. The implementation is available online[1].

4 Evaluation

To evaluate the impact of our construct on the performance of task parallel applications we have conducted several experiments based on our reference implementation. The results are discussed in the following subsections.

4.1 Fibonacci

Our first evaluation concludes our motivational example. In Sect. 2 various parallel implementations of `fib`, based on state-of-the-art parallel libraries and

[1] https://github.com/HerbertJordan/parec commit 9aa5dac.

language extensions, have been presented. All of them fall short in providing acceptable performance results for computing our benchmark case `fib(40)`. For a sequential execution time of ≈480 ms one would expect, presuming ideal scaling, an execution time of 40 ms on a 12-thread system, as illustrated in Fig. 1. Our `prec` based implementation obtains the result within 41 ms, corresponding to a 97.5% parallel efficiency. Clearly, our approach is able to mitigate the majority of overhead and to achieve acceptable performance for the given benchmark.

The evaluation of this fibonacci motivating example, as well as the comparison results shown in the motivation section, have been carried out using GCC 5.3.0, on a 6-core/12-thread Intel Core i7-5820K CPU at 3.3 GHz.

4.2 The `prec` Impact

In our second experimental setup, our goal was to identify the impact of considering the calling context like it is done by our `prec` operator compared to a purely localized task-generation code realized by utilizing a conventional `async` call. To eliminate any impact of the quality of an underlying runtime system, we utilized the same runtime implementation for both situations. Thus, we can exclude the effects of different scheduling policies, task queue lengths, stealing policies, or task handling overheads. To that end we compared two slightly different versions of our reference runtime:

- the `parec::async` configuration, where every call to `spawn` is treated like a `std::async` call, creating a new light-weight task to be scheduled by the runtime system
- the `parec::prec` configuration, where nested recursive tasks and their calling context are treated as described in Sect. 3

To compare those two setups, we ported the INNCABS [12] benchmark suite to the `prec` operator. The port can be obtained online[2].

Table 1 enumerates the eight benchmarks we have ported for our evaluation. The covered codes reach from numeric algorithms like Strassen and SparseLU, over combinatorial problems including QAP and NQueens, to standard utility algorithms like Sort. Table 1 also lists the problem sizes for our experiments. For practical reasons we decided to cancel all unfinished executions after 100 s and consider these to have timed out.

Our evaluation platform is a quad-socket shared-memory system equipped with Intel Xeon E5-4650 processors, each offering 8 cores clocked at a nominal frequency of 2.7 GHz (up to 3.3 GHz with Turbo Boost). The software stack consists of GCC 5.2.0 with -O3 optimizations, running on a Linux operating system with kernel version 2.6.32-473. The thread affinity for all benchmark runs was fixed using a fill-socket-first policy, and all reported numbers are medians over ten runs.

Figure 2 illustrates our experimental results. For each benchmark we show the median execution time for a varying number of threads. The execution times

[2] https://github.com/PeterTh/inncabs/tree/parec_port.

Table 1. Ported INNCABS benchmarks and problem sizes

Name	Description	Problem size
Fib	Fibonacci number	47
Health	Health care simulation	medium.input
NQueens	The N-Queens problem	13
Pyramids	2D cache-oblivious stencil solver	-
QAP	Quadratic assignment problem	chr15c.dat
Sort	Merge-sort	10^8 8192 2048 128
SparseLU	LU factorization	-
Strassen	Strassen algorithm	4096

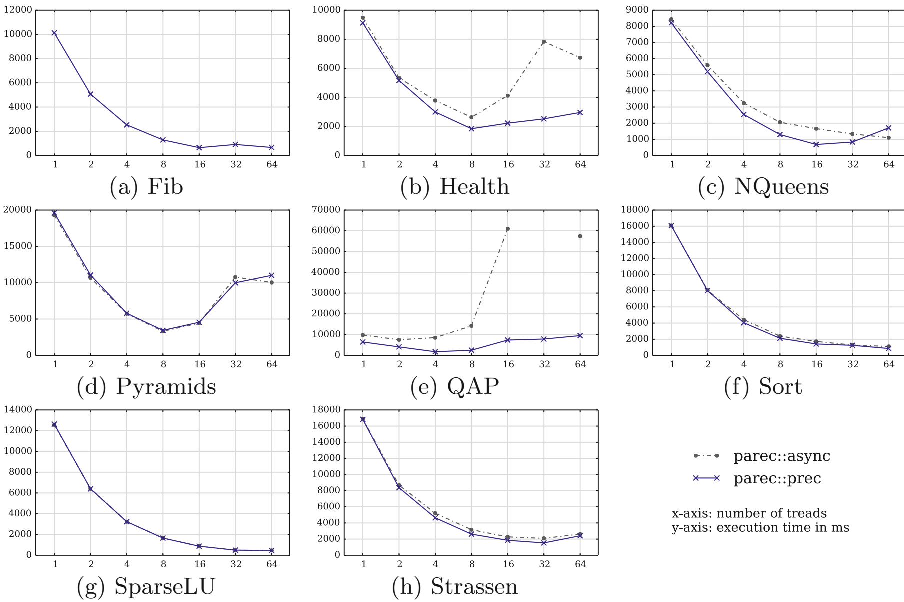

Fig. 2. Performance comparison of the `parec::async` and `parec::prec` operator mapped to the same runtime system.

for running in `parec::async` and in `parec::prec` mode are compared – with the exception of Fib, where the total run time of the `parec::async` configuration did not finish within a timeout of 100 s for any number of threads.

For benchmarks where task creation and spawning is clearly dominated by the actual workload of the task, no significant difference between the two configurations can be observed. This includes Pyramids, Sort, SparseLU, and Strassen. For all of those, both configurations produce almost identical results.

Benchmarks where the actual workload is small compared to the task spawning overhead however, benefit from context-awareness and specialization conducted by the `prec`. Those include

- Fib – with only one comparison, two function calls, and three arithmetic operations per task – here `prec` is faster by several orders of magnitude
- Health – where each task is a local operation on a single node in a graph
- NQueens – where each task conducts a small number of stack local operations
- QAP – where each task conducts stack local operations and a single access to a globally shared scalar value

Especially those benchmarks with a large, yet irregular fan out (Health and QAP) produce a large number of tasks in lower levels of the execution tree, the overhead of which can be significantly reduced by the `prec` approach.

For some of the benchmarks, increasing the number of threads beyond a single socket – thus exceeding 8 threads – shows a considerable change in their scaling behaviour. This is particularly prominent for Health, and QAP, in which each task accesses a single, globally shared scalar due to the branch-and-bound nature of the represented algorithms. Also, for some data intensive benchmarks like Pyramids, the data access order – and thus the efficient usage of caches – has a much higher impact on the execution performance of the benchmark than the task scheduling overhead. However, our reference runtime has not been specifically tuned to deal efficiently with this kind of challenges, which constitute large branches of research on their own.

As the data shows, the utilization of `prec` can provide substantial performance benefits, in particular for use cases with a low number of operations per tasks.

The raw data of this experiment, as well as all the sources and scripts used for their generation can be obtained online[3].

4.3 Application Benchmarks

For our final evaluation, we are comparing the absolute performance of our `prec` implementation with the performance obtainable by utilizing comparable parallel libraries – in particular GCC's Cilk Plus and `std::async`.

While Cilk Plus does not require additional tuning parameters, `std::async` does allow the user to specify a launch policy. According to the C++ standard, the following policies need to be supported:

- *async* – the spawned task is processed asynchronously
- *deferred* – the task is processed by the thread requesting the result (lazy evaluation)
- *default* – which is equivalent to either async or deferred, but leaves the choice to the implementation

For our comparison we evaluated all three of those policies and included the one providing the best performance for the respective number of threads. Furthermore, we utilized the same benchmarks and setup as in Subsect. 4.2.

[3] https://github.com/PeterTh/inncabs/tree/parec_port commit b3f87a2.

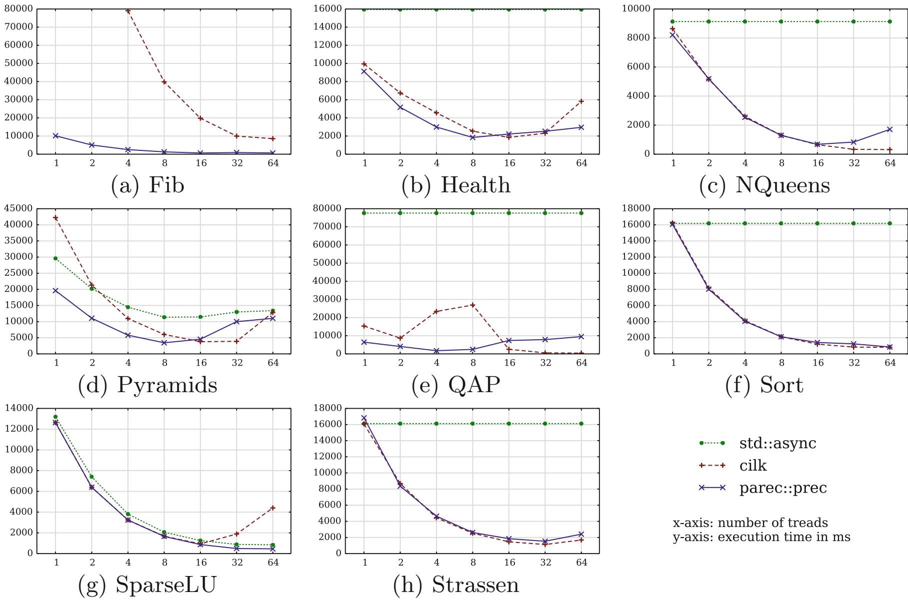

Fig. 3. Performance comparison of our `parec::prec` implementation, Cilk Plus, and C++'s `std::async`.

Figure 3 illustrates the obtained results. Similarly to our `parec::async` setup, `std::async` is not able to compute a solution for Fib within our time limit of 100 s.

In several cases the launch policy *deferred* turned out to be the fastest option, although leading to an effective serialization of the program code. As a result, for a set of benchmarks including NQueens, Sort, and Strassen, the execution time of `std::async` does not change with the number of threads.

Cilk Plus, on the other hand, also fails to obtain a result for Fib with only 1 or 2 threads within 100 s.

For NQueens, Sort, SparseLU, and Strassen, Cilk Plus and our reference runtime provide comparable results, while C++'s `std::async` only manages to obtain a speedup for SparseLU.

In particular for Fib, but also for the intra-socket QAP, our approach provides (highly) superior performance compared to the alternative libraries and/or language extensions.

For Pyramids, the Cilk and `std::async` implementation suffer from high sequential overhead due to their internal operation, which only Cilk manages to compensate due to almost linear scaling. However, within a single socket, our runtime manages to provide higher performance than both of them. Beyond the single-socket boundary, however, impacting factors like the evaluation order of sub-tasks and their effect on cache usage and NUMA effects become more important, weakening the performance of our runtime. However, those have not been

the objectives of the presented work. These aspects will be further investigated by follow-up efforts.

The raw data for this experiment, as well as full sources and scripts to reproduce it, may be obtained online[4].

5 Related Work

Due to its status as a fundamental and easy to use parallel abstraction, there is a large body of existing work in optimizing task parallelism. Particular attention was previously paid to scheduling strategies [2,9] and alleviating task creation overhead [5,10]. What is common to all of these approaches is that they focus primarily on the runtime system level, while we employ C++ template programming in order to introduce a code-generation component evaluated at compile time. This allows us to generate more efficient parallel code, and to provide any given runtime system with the option of switching to a zero-overhead sequential implementation without requiring the user to manually create and maintain separate versions of their code. As such, our approach is orthogonal to and compatible with any further runtime-level adaptation and optimization – such as the lazy task creation scheme described by Duran et al. [5] or any of the hardware-aware or locality-based scheduling strategies [7].

Looking specifically at the C++ language, parallelism is primarily the domain of traditional tasking libraries [1,11], which are also inherently limited to runtime optimization due to the type of primitives they offer. Meanwhile, current compiler research related to C++11 parallelism has focused on the correctness of the memory model underlying the standard [4], not on the performance of its library function implementations. An exception is the authors' previous work on semantics-aware compilation techniques [13], however, unlike the template library based approach presented in this paper it requires a non-standard system software stack, limiting its applicability in real-world software deployments.

Existing C++ template libraries for parallelism which operate on a higher level of abstraction, such as Quaff [6], aim to support a wide variety of parallel patterns. However, unlike the work presented in this paper, they do not focus specifically on reducing overheads in recursive task parallel algorithm by compile-time multiversioned code generation.

6 Conclusion and Future Work

In this paper we presented a novel, abstract parallel construct enabling parallel task library implementations to grasp and manipulate the context of task spawning points. By utilizing the capabilities established by its design, we demonstrated its potential of reducing the task creation overhead within nested recursive parallel codes. Our reference implementation generally achieved comparable

[4] The files for the `std::async` reference implementation, the Cilk Plus port, and the `parec::prec` port are available at `https://github.com/PeterTh/inncabs/tree/[master,cilk_port,parec_port]` respectively (commits a87ed27, 6476d75 and b3f87a2).

or better performance than state-of-the art solutions. Crucially, for a class of use cases in which the computational effort of individual tasks is low, our approach was able to attain superior performance. Furthermore, unlike the best state of the art competitor (Cilk Plus), which depends on compiler extensions, our approach is a pure C++14 solution and thus portable to any compliant compiler.

Due to its library based nature, our approach is easy to customize e.g. in its scheduling and version selection policy. More sophisticated concepts for these will be investigated to improve the load balancing and scalability of our runtime implementation. Furthermore, additional high-level parallel constructs including parallel loops, stencils, or map-reduce like operators can be designed on top of `prec` to improve its usability.

Acknowledgement. This project has received funding from the European Union's Horizon 2020 research and innovation programme as part of the FETHPC AllScale project under grant agreement No. 671603.

References

1. An, P., et al.: STAPL: an adaptive, generic parallel C++ library. In: Dietz, H.G. (ed.) LCPC 2001. LNCS, vol. 2624, pp. 193–208. Springer, Heidelberg (2003). doi:10.1007/3-540-35767-X_13
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency Comput.: Pract. Exp. **23**(2), 187–198 (2011)
3. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The design of openMP tasks. IEEE Trans. Parallel Distrib. Syst. **20**(3), 404–418 (2009)
4. Batty, M., Memarian, K., Owens, S., Sarkar, S., Sewell, P.: Clarifying and compiling C/C++ concurrency: from C++11 to power. In: ACM SIGPLAN Notices, vol. 47, pp. 509–520. ACM (2012)
5. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008, pp. 1–11. IEEE (2008)
6. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: Quaff: efficient C++ design for parallel skeletons. Parallel Comput. **32**(7), 604–615 (2006)
7. Guo, Y., Zhao, J., Cave, V., Sarkar, V.: SLAW: a scalable locality-aware adaptive work-stealing scheduler. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–12. IEEE (2010)
8. Jones, S.: Introduction to dynamic parallelism. In: GPU Technology Conference Presentation, vol. 338 (2012)
9. Lakshmanan, K., Kato, S., Rajkumar, R.: Scheduling parallel real-time tasks on multi-core processors. In: 2010 IEEE 31st Real-Time Systems Symposium (RTSS), pp. 259–268. IEEE (2010)
10. Mohr, E., Kranz, D.A., Halstead Jr., R.H.: Lazy task creation: a technique for increasing the granularity of parallel programs. IEEE Trans. Parallel Distrib. Syst. **2**(3), 264–280 (1991)
11. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media Inc., Sebastopol (2007)

12. Thoman, P., Gschwandtner, P., Fahringer, T.: On the quality of implementation of the C++11 thread support library. In: 2015 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 94–98. IEEE (2015)
13. Thoman, P., Moosbrugger, S., Fahringer, T.: Optimizing task parallelism with library-semantics-aware compilation. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015. LNCS, vol. 9233, pp. 237–249. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48096-0_19