On the Quality of Implementation of the C++11 Thread Support Library

Peter Thoman, Philipp Gschwandtner and Thomas Fahringer
Distributed and Parallel Systems Group
University of Innsbruck
Innsbruck, Austria, Technikerstrasse 21a
Email: {petert,philipp,tf}@dps.uibk.ac.at

Abstract—**Providing standardized building blocks for task-parallel programs within a language and its standard library has several advantages over other solutions. Close integration with compilers and runtime systems allows for potentially higher performance and portability facilitates wide-spread use.**

In the recently ratified C++11 standard, language constructs have been added along with a memory model to provide the developer with such building blocks. They allow accessing task parallelism and synchronization in a flexible and standardized way, potentially removing the need for third-party solutions. Nevertheless, since parallelization aims at high performance, an examination of the quality of implementation of these standardized means is necessary to determine their suitability for replacing established solutions.

To that end, we present INNCABS, a new cross-platform cross-library benchmark suite consisting of 14 benchmarks with varying task granularities and synchronization requirements. Based on these benchmarks, we demonstrate that the performance of C++11 parallelism constructs in the three most commonly employed C++ runtime libraries prevents their use as a full replacement for third-party solutions due to simplistic parallelism implementations and high synchronization overheads.

I. Introduction

Task-based parallelism is a widespread and useful paradigm, which has applications in areas ranging from embedded systems, over user-facing productivity software, to high performance computing clusters. The C++ programming language is one of the most widely-used languages for performance-sensitive applications in all of these fields. Before the introduction of the ISO/IEC C++11 standard [1], there was no way within the language or its standard library to target thread-level parallelism. This led to a proliferation of third party solutions for task parallelism in C++.

In the C++11 standard, which is now implemented in all the most widely-used C++ compilers, a memory model for parallel systems was specified in the base language [1], and based on this foundation several parallelism-related functions and classes were introduced in the standard library. One of the most interesting from both the perspective of an application developer and a library implementation is the `async` function template. It has the potential to express both coarse and fine-grained task parallelism, and can serve as a building block for more complex and feature-rich parallel patterns.

Clearly, providing a standardized building block for task parallelism has many advantages over a smorgasbord of third-party and homegrown solutions: it is easier to teach and read, thereby increasing programmer productivity, it can be more closely integrated and supported within a given compiler and its associated runtime library, thereby potentially offering superior performance, and it is portable to any standard-conformant implementation of C++ without external dependencies. However, the primary reason for parallelization is generally the desire to improve program performance. As such, the standard library facilities for parallelism need to provide a high *quality of implementation* in terms of runtime efficiency, overhead and scalability. Unlike traditional sequential algorithms, where specific complexity classes for time and space are mandated within the standard, doing the same for parallel programs is significantly more challenging. Therefore, we propose a new set of task-parallel benchmarks, INNCABS (Innsbruck C++11 Async Benchmark Suite), designed to evaluate the parallel performance of C++11 implementations over a variety of testing scenarios.

The concrete contributions of this paper are as follows:

- A new suite of 14 task parallel benchmarks, implemented in standard C++11, which feature distinct parallel patterns.
- An evaluation of the performance, overheads and scalability of the three most common C++11 library implementations on these benchmarks.

Section II will provide an overview of related work, followed by Section III introducing the C++11 `async` primitive and some related library functionality. The proposed INNCABS benchmark suite and each of its test cases are introduced in Section IV, and its results are used to evaluate the behavior and quality of implementation of existing C++ standard libraries in Section V. It is followed by a conclusion in Section VI.

II. Related Work

Three established areas of scientific and engineering work are relevant for this paper: task-parallel *language extensions*, parallel C and C++ *libraries* and *benchmark suites*.

In terms of languages and language extensions, Cilk was one of the first to establish a simple but reasonably complete set of parallel language primitives as well as providing a high-quality implementation and runtime system based on work-stealing, later refined and extended for C++ in Intel's Cilk Plus [2]. The widely used OpenMP standard for shared-memory multiprocessing added task support in version 3.0 of the standard [3]. Finally, a more research-oriented platform is provided by StarSs [4], allowing to target also distributed memory clusters and accelerators with task constructs. All of these extensions provide unique features and solid implementations, but they lack in portability, maintainability and ease of deployment in comparison to functionality included directly in the C++ standard, such as the *async* primitive.

Many C and C++ libraries provide support for task-based parallelism. Perhaps the most widely used of these are the Intel Threading Building Blocks (TBB) [5], a rich, portable, and mature platform for shared memory parallel systems. The Boost [6] project *Thread* component provides an `async` primitive very similar to the standardized C++11 option, as well as a range of experimental *Executors* which implement concepts with a potentially high performance impact, such as thread pools. The HPX project [7] is more ambitious and more research-focused, as it also targets distributed memory. While such libraries are capable of providing good parallel performance, none of them are as ubiquitous, easy to integrate and deploy and, hopefully, well-tested as the standard library. As such, we believe that they cannot replace it. Crucially, their existence does not justify a lack of performance and scalability testing in the standard library primitives, but they should rather be used to provide a guideline regarding the performance to be achieved.

Finally, a large number of established parallel benchmark suites for C and C++ do already exist. However, the vast majority of these – including the popular SPEC OMP [8] and NAS parallel benchmark [9] suites – predominantly or exclusively feature a flat, loop-oriented parallel structure. Others, such as ParBenCCh [10] are based on either language extensions or outdated C++ concepts and libraries. Conversely, INNCABS is designed to measure the performance of a variety of *task-parallel* programs using *pure standard C++11*. A selection of the individual INNCABS benchmarks is based on adapted code from the Barcelona OpenMP Tasks Suite [11], the single most widely-used task-parallel benchmark suite, though many of these codes are in turn adapted from previous benchmarks originally written for Cilk.

III. Library Specification Overview

The C++11 *thread support library* provides classes and functions for dealing directly with threads, guaranteeing mutual exclusion, signaling conditions, and exchanging data between asynchronous tasks using futures. Of these facilities, we consider the `async` primitive and the variadic `lock` function for the deadlock-free acquisition of multiple mutexes the most interesting from a quality of library implementation perspective. Standard library threads generally map directly to operating system (OS) threads, and so do mutexes in a single-lock scenario. As such, their performance is influenced primarily by the underlying OS facilities, and beyond the control of the library implementation. For `async` and `lock`, however, the situation is different, and they are therefore covered in INNCABS. We will now provide a short overview of these primitives, and explain why they allow for highly significant implementation differences.

A. *The* `async` *Primitive*

In the C++11 standard, the async function template is defined as follows:

```
template< class Function, class... Args >
std::future< typename std::result_of<
            Function(Args...) >::type >
async( std::launch policy, Function&& f,
       Args&&... args );
```

Semantically, it launches a given function `f` with arguments `args` according to some `policy`, and returns an instance of the `std::future` template class which can be used to *asynchronously* query its result. The `policy` parameter is of type `std::launch`, and can be either `std::async`, `std::deferred`, some implementation-defined value, or any combination of those. The policy parameter is optional, and, crucially, if omitted it defaults to `std::async | std::deferred`. Throughout the remainder of this document, we will refer to these launch policies as *async*, *deferred*, and *optional*. These options are defined as follows:

- **async** always launches a *new thread* to execute the provided function asynchronously.
- **deferred** implements *lazy evaluation*, that is, `f(args...)` is executed synchronously the first time the returned future object is queried.
- **optional** gives the *implementation a choice* of whether to execute the given function asynchronously or synchronously.

Of these policies, *optional* is the most interesting in terms of implementation quality, as it gives the library implementation a large room for optimization. It is a well-studied fact that optional task parallelism is highly advantageous when managing the degree of software parallelism in a system, and can greatly influence parallel performance and scalability [12].

As such, it is a good design decision to make the optional policy the default, as it allows this type of optimization to be performed. Consequently, we will focus on the performance of this option during the evaluation in Section V, though results with all policies will be provided for completeness.

B. *The Variadic* `lock` *Function*

Unlike the `lock` method of the `std::mutex` class, which has clear semantics and generally maps directly to an OS primitive, the `std::lock` function provides more room for library implementations to influence performance. Syntactically, it is defined as follows:

```
template< class Lockable1, class Lockable2,
          class LockableN... >
void lock( Lockable1& lock1, Lockable2& lock2,
           LockableN& lockn... );
```

Its semantics are interesting, as the standard stipulates that it shall employ an (unspecified) deadlock avoidance algorithm in order to lock all the lockable objects passed to it. Clearly, there are many implementation options for this functionality, and their performance may differ significantly across platforms, usage scenarios and external load profiles. Therefore, INNCABS includes a benchmark specifically designed for testing the performance of this function, and it is also used heavily in another benchmark (intersim).

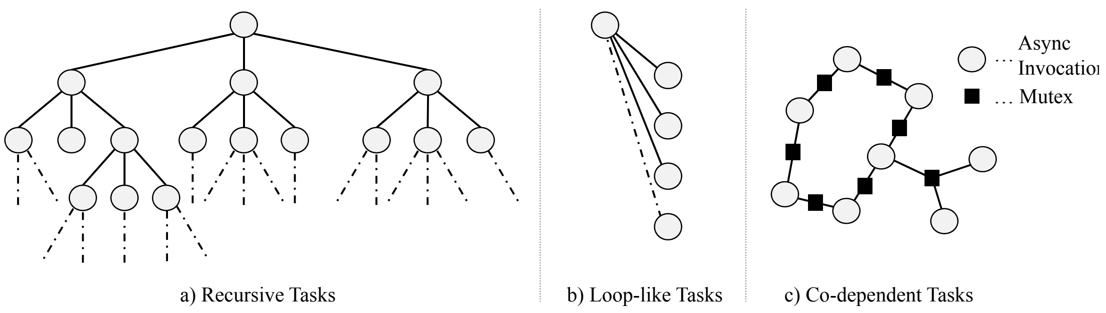

Fig. 1: Task-parallel structure types

IV. The INNCABS Programs

In this section we present the overall design of and the individual programs included in INNCABS. These explanations are intended to focus not on the actual algorithms or calculations the benchmarks perform, but rather on their *parallel structure*, their average *task granularity*, and their *synchronization requirements*. Knowing these features for each benchmark is essential in order to interpret their performance results on a given testing hardware/software platform.

Figure 1 illustrates the three categories of task-parallel structures which are featured in the benchmark suite. They can be summarized as follows:

- *Recursive* parallelism features nested `async` invocations forming a tree pattern. Up to some cutoff point, each `async` function spawns multiple further asynchronous calls. This is the typical structure traditionally encountered in Cilk programs, and common in functional programming languages. For this type of parallelism, the *arity* and *depth* are significant parameters influencing runtime behavior and performance.
- *Loop-like* structures simulate loop parallelism using the `async` primitive within a basic for or while loop running in a root thread. These are comparable to traditional loop parallelism, and present a very different scheduling challenge compared to recursively nested parallel programs. Therefore, a few such cases are included in INNCABS.
- *Co-dependent* task parallelism in this case refers to a sea of asynchronously spawned tasks, which depend on a set of mutexes shared by two or more participants. These benchmarks test the performance of the deadlock-free lock acquisition algorithm implemented in the `std::lock` function.

Note that some benchmarks may use combinations of the above, e.g. a nesting of multiple levels of loop-like parallelism.

As a benefit of being implemented within the INNCABS framework, each benchmark application has standardized access to the following features: (i) execute all `async` calls with any combination of `std::launch` parameters; (ii) built-in result testing for successful execution; (iii) support for multiple runs with statistical evaluation; (iv) multiple output formats, including comma-separated values (CSV); (v) timeout support in order to limit the execution time of complete benchmark runs to reasonable values (see Section V for details).

TABLE I: INNCABS benchmark characteristics

benchmark	origin	structure	granularity	synchronization
Alignment	AKM	loop-like	coarse	-
FFT	Cilk	rec. balanced	variable	-
Fib	-	rec. balanced	fine	-
Floorplan	AKM	rec. unbalanced	fine	atomic/pruning
Health	BOTS	loop-like	moderate	-
Intersim	New	co-dependent	fine	mult. mutexes per task
NQueens	Cilk	rec. unbalanced	moderate	-
Pyramids	New	rec. balanced	coarse	-
QAP	New	rec. unbalanced	fine	atomic/pruning
Round	Hinnant	co-dependent	fine	two mutexes per task
Sort	Cilk	rec. balanced	variable	-
SparseLU	BOTS	loop-like	coarse	-
Strassen	Cilk	rec. balanced	moderate	-
UTS	UNC	rec. unbalanced	variable	-

While a detailed description of each benchmark's characteristics can be found online[1], a summary of the applications and their characteristics is presented in Table I. The "origin" column contains the original source of the benchmarks (which were rewritten for C++11): "BOTS" refers to the Barcelona OpenMP Tasks Suite, "Cilk" to the Cilk language distribution, "UNC" to the University of North Carolina and "AKM" to the Cray Application Kernel Matrix. The benchmarks marked as "New" were created for INNCABS and are based on the respective algorithms mentioned in their description. If an application uses additional synchronization beyond the joining of `future` objects returned from `async` calls, its type is listed in the "synchronization" column. The INNCABS framework and all of the benchmark codes are available online[1].

V. Performance Results

A. Experimental Setup

Our target platform is a quad-socket shared-memory system equipped with Intel Xeon E5-4650 Sandy Bridge EP processors, each offering 8 cores clocked at a frequency of 2.7 GHz (up to 3.3 GHz with Turbo Boost), and a total of 256 GB of main memory. The software stack consists of clang 3.4.2 using libc++ 3.4.2 and -O3 optimizations (abbreviated as *clang* throughout the remainder of this section), gcc 4.9.0 using libstdc++ 3.4.20 and -O3 optimizations (*gcc*) on a Linux OS with a 2.6.32-431 kernel, as well as the MS Visual C/C++ compiler 2013 v120 using msvcp 12.0.21005.1 (*vc*) and -Ox optimizations on MS Windows Server 2012 R2.

The thread affinity for all benchmark runs was fixed using a fill-socket-first policy. Hyper-threading was enabled,

[1]https://github.com/PeterTh/inncabs

TABLE II: Overall results for *clang*, *gcc* and *vc*. ^{A} Variable performance due to early tree pruning. ^{B} Inefficient lock implementation. ^{C} Concurrency error in C++ runtime library.

benchmark	clang:async	gcc:async	vc:optional	vc:async
Alignment	scales to 64	scales to 64	scales to 64	scales to 32
FFT	timeout	partial timeout	no scaling	timeout
Fib	timeout	timeout	scales to 8	timeout
Floorplan	timeout	no scaling^A	partial timeout	timeout
Health	timeout	partial timeout	scales to 8	timeout
Intersim	timeout	timeout	scales to 32	timeout
NQueens	timeout	timeout	scales to 8	timeout
Pyramids	partial timeout	scales to 8	scales to 16	scales to 16, overhead^B
QAP	timeout	no scaling^A	scales to 8	timeout
Round	scales to 64	scales to 64	scales to 64, overhead^B	scales to 64, overhead^B
Sort	timeout	scales to 8	scales to 64	timeout
SparseLU	scales to 64	scales to 64	scales to 64	scales to 32
Strassen	timeout	timeout	scales to 4, partial error^C	timeout
UTS	timeout	timeout	scales to 16	timeout

but hardware threads sharing a core were only used when employing all threads of the system. Time was measured using `std::chrono::high_resolution_clock` and all reported numbers are medians over five runs. A timeout was set at 900 seconds in order to limit total execution time, as the exact amount of time beyond this degree of slowdown is not particularly informative – sequentially, all benchmarks complete within 60 seconds or less on all platforms.

B. General Observations

While in theory, the *optional* policy of the `async` primitive is supposedly the most interesting in terms of quality of implementation, our results show that it simply maps to *async* on *clang* and *deferred* on *gcc*. This impression, based on the obtained data, was confirmed by a study of the source code of both libraries. Hence, detailed presentation of these cases is omitted for brevity.

Table II presents the results for the *optional* and *async* cases of *clang*, *gcc*, and *vc*. In order to make more effective use of the available space, instead of providing all 4410 numbers obtained in our benchmark runs, we focus on a qualitative analysis of the overall performance of each implementation. As evident, *clang* shows poor behavior for all benchmarks except for *Alignment*, *Round* and *SparseLU*, which also scale well in *gcc*. Unlike *clang*, *gcc* also provides acceptable performance in the *Pyramids*, *Floorplan* and *QAP* benchmarks. The Microsoft *vc* compiler and its library is the only standard implementation to make use of the *optional* launch policy, allowing it to achieve better performance in the vast majority of INNCABS benchmarks using this setting.

Adhering to the categorization presented in Section IV, detailed discussion of the results is grouped into benchmarks featuring *recursive*, *loop-like*, and *co-dependent* parallelism, with a further division of the *recursive* category into *balanced* and *unbalanced* tree structures.

C. Recursive Parallelism

1) Balanced Tree Structure: Relatively coarse-grained, balanced recursive parallelism is an ideal case in terms of parallel scaling, and places the least burden on a given runtime library

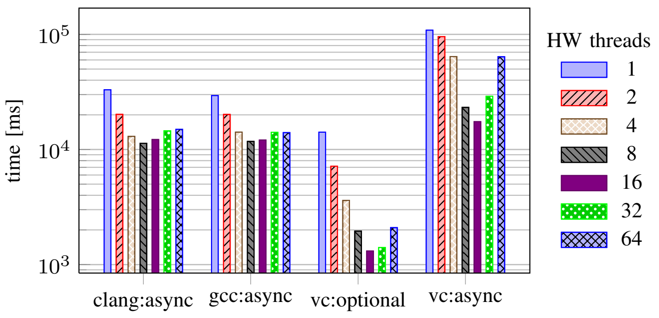

Fig. 2: Execution times for the *Pyramids* benchmark

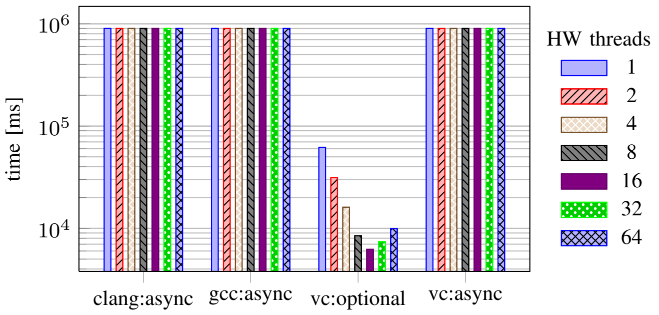

Fig. 3: Execution times for the *UTS* benchmark

implementation. The *Pyramids* benchmark represent such a use case within the INNCABS suite. Consequently, it scales to some extent in all tested implementations and launch types.

However, as shown in Figure 2, even for a relatively easy to scale case such as this, leveraging the opportunities offered by an optional launch policy greatly improves scalability and overall performance. The difference between the *vc:optional* and *vc:async* results is stark, and demonstrates both the power of optional parallelism and the high cost of OS-level thread creation on Windows.

Both the *clang* and *gcc* implementations perform comparably, and fall in between the two *vc* results. It is also noteworthy that *Strassen* failed for *vc:optional* when using 8 or more hardware threads with a `Concurrency::scheduler_resource_allocation_error` exception, indicating an issue in the msvcp concurrency library.

2) Imbalanced Tree Structure: The *UTS* benchmark represents an extreme case of fine-grained, highly imbalanced parallelism. As such, it challenges runtime systems in both their capabilities for low-overhead invocation or elision of parallel tasks, and in the dynamic redistribution of workloads across hardware resources.

As Figure 3 illustrates, none of the existing C++11 implementations manage to scale to the full degree of hardware parallelism in this case. In fact, all versions except for *vc:optional* time out, requiring far more time than a simple sequential execution of the benchmark. However, the Microsoft implementation of optional asynchronous launches manages to scale up to 16 hardware threads.

D. Loop-Like Parallelism

While loop-like parallelism cannot be directly represented using the C++11 thread support library, it is trivially ex-

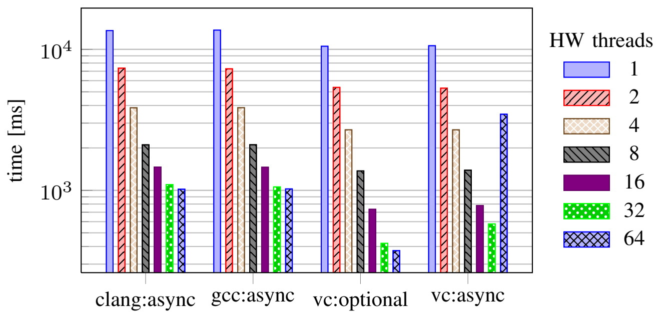

Fig. 4: Execution times for the *SparseLU* benchmark

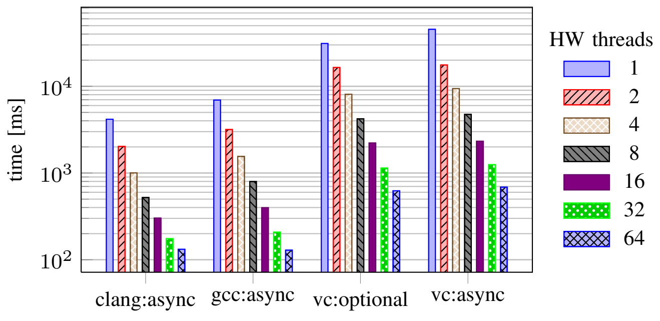

Fig. 5: Execution times for the *Round* benchmark

pressed using a composition of base-language loops and `std::async` calls for each iteration or set of iterations. This use case is most accurately represented by the *SparseLU* test case in INNCABS, which accordingly scales well on all implementations. Figure 4 summarizes the results obtained on each platform in this benchmark.

Beyond the generally good scaling behavior of the benchmark, two results stand out: *vc:optional* scales better beyond 8 threads than the *gcc* and *clang* alternatives, which is explained by its adaptive nature and resulting lower overhead per additional hardware thread. On the other hand, the 64 threads result for *vc:async* illustrates a common issue with allocating a large number of hardware threads on Windows OSes.

E. Co-dependent Parallelism

Figure 5 illustrates the results of the *Round* benchmark for each of our tested platforms. Both the *gcc* and the *clang* implementation perform very well in this oversubscribed locking scenario, while the results for *vc* indicate that significant overhead is introduced by the deadlock-free mutex acquisition method its library employs. While we cannot investigate the source code in this case, based on the results it seems likely that a loop based on repeated `try_lock` calls is used. Hence, when the degree of oversubscription is higher while being limited to a small number of hardware threads, there is a particularly large performance drop compared to the other implementations. Conversely, both *gcc* and *clang* use mechanisms which yield to the operating system when a locking attempt fails, and perform well.

VI. Conclusion

We have presented a new benchmark suite for C++11 parallelism, INNCABS, comprising 14 benchmark applications and covering three major types of task parallelism: loop-like, recursively nested and co-dependent. The implementation quality of the three predominant C++ standard library implementations (libstdc++, libc++ and msvcp) was evaluated for up to 64 hardware threads based on these benchmarks, and a number of shortcomings were identified.

While C++11 provides a programmer-friendly and flexible interface for task parallelism with the `std::async` function, our findings indicate that its current implementation quality across platforms prevents its intended use as a replacement for third-party or custom parallel task libraries. In particular, heavy performance degradation can be observed when fine-grained tasks are utilized, and in two of the three implementations (libc++ and libstdc++) no effort is currently made to implement any kind of adaptive decision making or low-overhead user-level threading in cases where this would be viable. Conversely, the Microsoft C++ library provides much better performance for fine-grained tasks, but employs a spinning mechanism in `std::lock` which heavily affects performance in oversubscription scenarios.

By providing a set of open source benchmarks (INNCABS) and an in-depth cross-platform and cross-library performance evaluation we hope to motivate the authors and companies behind each of these implementations to improve on their respective weaknesses, and make the C++11 standard parallelism functions a viable choice for cross-platform high performance applications.

Acknowledgment

This research has been partially funded by the Austrian Research Promotion Agency under contract 834307 (AutoCore) and by the FWF Austrian Science Fund under contract I01079 (GEMSCLAIM).

References

[1] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012.

[2] A. D. Robison, "Cilk plus: Language support for thread and vector parallelism," Talk at HP-CAST 18, 2012.

[3] E. Ayguadé, N. Copty, A. Duran *et al.*, "The design of openmp tasks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 3, pp. 404–418, 2009.

[4] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[5] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[6] B. Schling, *The boost C++ libraries*. Xml Press, 2011.

[7] M. Anderson, M. Brodowicz, H. Kaiser, and T. L. Sterling, "An application driven analysis of the parallex execution model," *CoRR*, vol. abs/1109.5201, 2011. [Online]. Available: http://arxiv.org/abs/1109.5201

[8] V. Aslot, M. Domeika, R. Eigenmann *et al.*, "Specomp: A new benchmark suite for measuring parallel computer performance," in *OpenMP Shared Memory Parallel Programming*. Springer, 2001, pp. 1–10.

[9] D. H. Bailey, E. Barszcz, J. T. Barton, *et al.*, "The nas parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[10] B. Carnes. (2001) The ParBenCCh 1.0 benchmark code. [Online]. Available: http://asc.llnl.gov/computing_resources/purple/archive/benchmarks/parbencch/

[11] A. Duran, X. Teruel, R. Ferrer *et al.*, "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*. IEEE, 2009, pp. 124–131.

[12] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *High Performance Computing, Networking, Storage and Analysis, 2008. International Conference for*. IEEE, 2008, pp. 1–11.