The Journal of Supercomputing (2024) 80:13714–13737
https://doi.org/10.1007/s11227-024-05987-0

# A new thread-level speculative automatic parallelization model and library based on duplicate code execution

**Millán A. Martínez**[1] · **Basilio B. Fraguela**[1] · **José C. Cabaleiro**[2] · **Francisco F. Rivera**[2]

Accepted: 9 February 2024 / Published online: 11 March 2024


## Abstract

Loop-efficient automatic parallelization has become increasingly relevant due to the growing number of cores in current processors and the programming effort needed to parallelize codes in these systems efficiently. However, automatic tools fail to extract all the available parallelism in irregular loops with indirections, race conditions or potential data dependency violations, among many other possible causes. One of the successful ways to automatically parallelize these loops is the use of speculative parallelization techniques. This paper presents a new model and the corresponding C++ library that supports the speculative automatic parallelization of loops in shared memory systems, seeking competitive performance and scalability while keeping user effort to a minimum. The primary speculative strategy consists of redundantly executing chunks of loop iterations in a duplicate fashion. Namely, each chunk is executed speculatively in parallel to obtain results as soon as possible and sequentially in a different thread to validate the speculative results. The implementation uses C++11 threads and it makes intensive use of templates and advanced multithreading techniques. An evaluation based on various benchmarks confirms that our proposal provides a competitive level of performance and scalability.

**Keywords** Speculative parallelism · Automatic parallelization · Thread-level speculation · Template metaprogramming

✉ Millán A. Martínez
millan.alvarez@udc.es

Basilio B. Fraguela
basilio.fraguela@udc.es

José C. Cabaleiro
jc.cabaleiro@usc.es

Francisco F. Rivera
ff.rivera@usc.es

[1] Universidade da Coruña, CITIC, Computer Architecture Group, 15071 A Coruña, Spain

[2] Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Dpto. Electrónica e Computación, Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain

## 1 Introduction

Multicore processors have become widely available nowadays. Although single-core performance continues to improve, this progress has slowed down, and due to physical and technological limitations, it is increasingly difficult to increase the clock speed of processors significantly. Computer architects decided to package multiple computing cores together on a single chip, and this has intensified over the years, with processors with a larger number of processing cores becoming more common.

These new multicore architectures make it possible, and even necessary, to exploit parallelism in several ways. To speed up a single process, it must be specifically programmed to take advantage of the parallel resources offered by the system. Compilers are often unable to extract parallelism hidden in sequential code, so it is not always possible to automatically take advantage of the parallel computing power of new systems, and developers must perform this process manually.

In recent decades, numerous languages, tools and libraries have been developed to alleviate this inconvenience, making it easier for programmers to take advantage of parallel capabilities. OpenMP in shared memory and MPI for distributed memory systems stand out. However, writing parallel programs is still a complex and error-prone task, requiring a deep knowledge of the code, the programming model to be used and the system's architecture. This makes very attractive the idea of being able to parallelize code automatically. In this way, once a potentially parallelizable section has been identified, usually a loop, a tool or library with automatic parallelization capabilities can either apply a general parallelization technique to the code or analytically extract its intrinsic parallelism.

In this context, speculative parallelization includes techniques that optimistically execute parallel loops that cannot be parallelized at runtime because of dependencies. It incorporates checking mechanisms to ensure the validity of the results and recovery mechanisms to recover and resume the execution in case of failed speculation. These techniques can be implemented at the hardware or software levels and at different layers. One of the most interesting software-level options is thread-level speculation (TLS), where the execution will be performed in several threads simultaneously, and these will implement different strategies. One of the main drawbacks of TLS techniques is the high computational cost required by the checking and tracking. Note that both must be carried out to detect when dependency violations have occurred on the speculative data shared by threads so that corrective actions can be taken. Our new proposed model presented in this paper follows an alternative speculative strategy: at the same time that a section of the sequential program is executed in a processor core, the speculative execution of an automatically parallelized version of it is also launched in the free cores. If, at the end of the execution, the speculation was correct, the results were be obtained in less time. In the case of a failed speculation, the valid results are obtained from the sequential version, and the results of the failed speculative parallelization are discarded. This model allows nested speculations of several sections, in such a way that speculative results from a previous section can be used as a starting point for another speculative process in the next section.

Speculative parallelization has several advantages, among which it stands out that it can be applied to practically any loop even if it is not initially parallelizable, either because it is not analysable at compile time or because there are potential dependencies, race conditions or indirections that prevent a traditional conservative parallelization. The main limitation of speculative parallelization is that the potential parallel performance that can be obtained is less since you either need to introduce performance-intensive code to detect and handle the dependencies or run the code in duplicate, as in the case explained above.

This paper also presents a new high-performance library developed in C++ called `SpecLib`, which allows automatic speculative parallelization of code based on C++11 threads implementing our new proposed model. The library aims to be easy to use for users, who only have to identify each potentially parallelizable section of the code and encapsulate it in a function that will be passed in the call to the library, requiring very few modifications to the original code. The C++ language has great advantages thanks to its templates, offering great flexibility and allowing the use of the library with a wide variety of code types and with a different number of parameters and configurations while maintaining optimal performance since templates are applied at compile time. In addition, the C++ threads are available on almost all platforms, and the library only uses elements of the standard library that are also available on all platforms that support C++14. The implementation and some examples will be made publicly available upon acceptance of this manuscript at https://github.com/UDC-GAC/speclib.

The rest of this manuscript is organized as follows: Sect. 2 reviews the related work and Sect. 3 details the new proposed TLS model, while Sect. 4 presents the new `SpecLib` library, its syntax and implementation. The performance evaluation is found in Sect. 5. This is followed by our conclusions and future work in Sect. 6.

## 2 Related work

While there is a large variety of approaches for thread-level speculation [19], these techniques are almost always only applied to loops, as they are the natural source of work distributable among threads. Another common point is the usually significant overhead of these techniques because of the cost associated with detecting dependency violations, maintaining current and past state data, the related recovery and correction mechanisms after a failed speculation, the synchronization and communication between threads and the potential load imbalance and poor cache usage due to data sharing [8]. Our proposal reduces many of these overheads by simplifying some of these mechanisms, but in exchange, it requires executing the code twice, both sequentially and in parallel. As shown in Sect. 5, the performance obtained with this approach is, for some instances, considerably better than continually using complicated and expensive dependency violation analysis and correction techniques.

The `ATLaS` project [1] adds speculative parallelization support for the C language through either a library or a compiler plugin that extends OpenMP to use this library through a directive. Their approach is totally different from ours. Namely, `SpecLib` executes the code not only in parallel to get speculative values

prematurely but also sequentially to validate these results. However, ATLaS bases its validation on dependency violation detection techniques. It keeps track of the data to know when it is written or read, so that it can detect if the speculative optimistic version of these data must be invalidated. We consider that ATLaS is one of the most modern and performant libraries that apply thread-level speculative parallelization in loops. For this reason, it will be the reference to validate our proposal in Sect. 5.

Going back in history, one of the first software solutions for thread-level speculation in loops is [14], which speculatively transforms and launches the loop while a test is applied to verify that no dependency violation occurs. If the test fails, the loop is executed again sequentially. An evolution of this solution [4] extends this technique to loops that are known to have certain dependencies. Later, [22] introduces the idea of using a master/slave model. The program will be divided into different tasks executed by the slaves. The master thread will try to predict the final result of each task and will continue its execution speculatively without waiting for its results. This approximation must be checked as soon as possible to verify whether the speculative prediction succeeded. In case of failure, the execution will be restarted from the last checkpoint.

A different approach is developed in [3], where an aggressive sliding window is used to avoid as much as possible having to synchronize the threads while checking in all the store operations that a dependency violation does not occur. DSWP [11] attempts to take advantage of the fine-grained pipeline parallelism of some applications to automatically extract long-running concurrently executing threads, while [20] combines profile-driven parallelism detection with machine learning to identify and maximize the number of loops that can be safely parallelized and to make better mapping decisions for different target architectures, although it requires user support for final approval.

Fast Track [5] divides the code into two versions: the fast insecure track and the normal track. The fast one is speculatively executed using unsafe optimizations of the sequential code, while the normal track allows checking the results. In [17, 18], the state of speculative parallel threads is maintained separately from the non-speculative computation state. The speculative execution is performed in three sections: prologue, speculative, and epilogue. The prologue contains the instructions that could not be speculatively executed. The speculative section includes the parts that are not expected to suffer a dependency violation. Finally, the epilogue is intended to manage output data.

The compiler framework for thread-level speculation in [2] relies on compile-time directives that allow parallelizing all the code instructions, not just those in loops. The user first needs to analyse and profile the code to produce a control flow graph in order to execute speculative parallel code on multiple threads. The Mitosis compiler framework [12] tries to predict the thread's input values based on pre-computation slices, which are built and added at the beginning of the speculative thread. These pre-computed slices must compute thread input values accurately, but they do not need to guarantee correctness since the underlying architecture can detect and recover from misspeculations. This allows the compiler to use aggressive and unsafe optimizations.

The `Galois` model [6, 7] provides programming abstractions that allow exploiting the parallelism of sequential programs by a runtime system that uses specific hints to speculatively execute code in parallel, using locks to ensure data consistency. Inverse operations are used to recover in a failed speculation following an undo log defined for each iteration.

The `Speculative Memory (SM) TLS system` [10] proposes a scheme based on shelving code blocks to keep a correct order of dependent executions while allowing concurrent non-dependent executions. This removes some types of inter-iteration dependencies and avoids using synchronization points.

In [9], a new architectural framework and speculative algorithm is presented, especially designed for intelligent systems and to take advantage of the Hardware Transactional Memory (HTM) instructions available in some processors. The framework describes three phases. The first one converts the code to an LLVM intermediate code. The second one analyses hotspots through profiling to evaluate the best sections to optimize. Finally, a memory and dependency analysis determines how the multithreaded speculative code is transformed and generated.

Another proposal that takes advantage of HTM support is [15], where a new clause for OpenMP called `tls` is presented. This is applied to the already existing `taskloop` clause to allow speculative parallelization in loops with dependencies. In [16] preprint, its functionality is extended and evaluated in more detail.

`Speculative Lock Elision` [13] automatically replaces parallel code locks with optimistic hardware transactions, expecting no errors. If transactions fail, the original locks must be used. Some solutions, such as those in [21] combine several already known solutions, such as helper threads and run-ahead techniques with thread-level speculation to achieve a suitable combination that improves the final performance.

Finally, it is worth mentioning that there are also purely hardware-based approaches, although they are out of the scope of our work.

## 3 A new model for thread-level speculative parallelization of loops

The first step of the model consists in dividing the total set of iterations of the loop to parallelize into several chunks of iterations or blocks. These chunks will be defined either by their size or by the total number of blocks, whose value is usually established by the user.

One of the main challenges of the different TLS algorithms is to keep track of the speculative data shared between threads to detect when a dependency violation occurs on them and to be able to take the appropriate actions. This is usually very expensive in terms of performance. The new speculative parallelization model that we present aims to avoid having to keep track of the state of speculative variables. To achieve this, the code will be executed in duplicate. A speculative parallel version will initially optimistically assume that it was executed correctly, and a sequential version ensures correct results and allows the validations of the speculative ones.

The planning and execution of the blocks of iterations will be established in an orderly manner since to start the execution of a block, the execution of the previous

block must have finished. When the execution of a block is scheduled, the number of threads available by the system is checked, as it can vary. In one of the available threads, a sequential execution of the block will be launched. Simultaneously, in the rest of the available threads, the execution of the same block will be launched in parallel, distributing the iterations of that block among the different threads available. Therefore, there are two simultaneously running versions of the same block, a sequential execution in one thread and a parallel execution spread across multiple threads. The sequential execution will always get correct results if it initially starts from correct values. The parallel execution may result in correct or incorrect values, but this will not be known until they can be compared against the values obtained in the sequential version. However, as soon as the parallel version finishes, the model will use its computed result values to start the execution of the next block, and it will be done even though, at this point, it is not yet known if these calculated values are correct or incorrect. When the sequential version finishes, a validation will be performed to check whether the values obtained earlier in the parallel version are valid. If they are not, all subsequent block executions that started from these values must be cancelled since they started from incorrect values and, therefore, their calculations are also incorrect. Note that the execution is resumed from the last checked good values. In this case, no time has been gained, but neither has it been lost in relation to a sequential execution. However, if the validation was correct, that means that the calculations that were started early from these speculative values, now verified as correct, are valid. This early time is the time gain obtained in the speculation.

Figure 1 shows a simple example of the execution of the first three blocks of a loop using four threads. All four threads are available for the first block `b1`. In this case, one of the threads, *T1*, is in charge of executing the sequential version `b1s`. The three remaining available threads will execute the parallel version `b1p`, dividing the work among them. When this parallel version ends, the model optimistically assumes that the speculative execution was correct, and therefore proceeds to execute the second block `b2` based on the results obtained speculatively in the parallel version of the first block. Since there are three threads available at this point, this second block is launched sequentially as `b2s` in *T2*, while in the

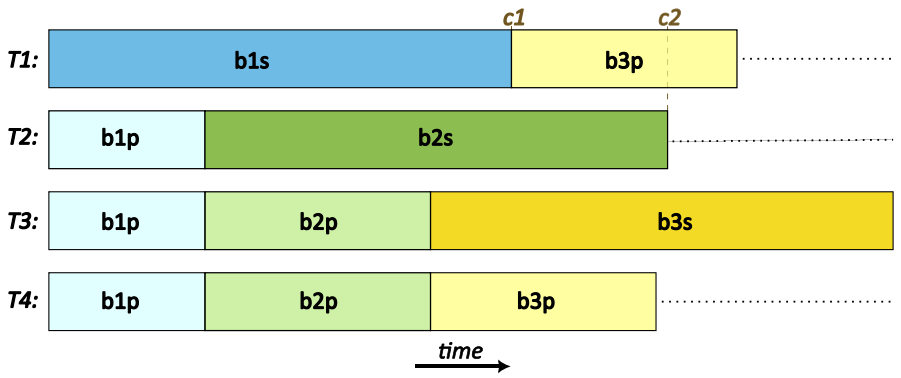



**Fig. 1** Speculative execution of the first three blocks of a loop using four threads

remaining two threads, the parallel version `b2p` is executed. This parallel version of the second block ends even before the sequential version of the first block has finished. In this moment only the third block remains to be executed, and just two threads are available. In one of them, the sequential version `b3s` is executed as always, while for the parallel version `b3p` only one thread is available. The minimum number of threads on which the parallel version can be planned is *two* threads for this example, although this minimum could be adjusted to any desired value. Therefore, the parallel version of the third block is planned for two threads, so that in the current available thread, the first section is executed, while the other section will be executed as soon as a new thread is available.

In this example, the checks are carried out in `c1` for the first block as soon as its sequential version `b1s` ends in *T1*, and in `c2` for the second block when its sequential version `b2s` ends in *T2*. In this case, both checks are correct. The speculation of the blocks is multilevel and dependent on the previous blocks since a block can start from speculative values calculated in the previous block and so on. Therefore, it can be planning and executing a new block of iterations, although the result values of previous blocks have not yet been validated. In this example, note that even if the second and third blocks validate successfully, if the first block had a failed validation, their work would have been discarded since they all depend on the speculative values calculated in advance in the parallel version of the first block.

The consequences of a failure in the speculation of one of the blocks are depicted in Fig. 2, where the failed validation is marked as *f3*. It shows a situation in which at the end of the sequential execution of block number three `b3`, it is verified that the calculated parallel value is invalid, so all subsequent executions are cancelled and their results invalidated, recovering and resuming execution with the block following the one just checked, the fourth block `b4`.

Figure 3 shows the worst case, in which all blocks fail the check once their sequential version ends. The total execution time is almost the same as the one resulting from a sequential execution, regardless of overheads, which may become relevant, especially for recovery stages such as those needed after the failed validations in `f1` and `f2`.

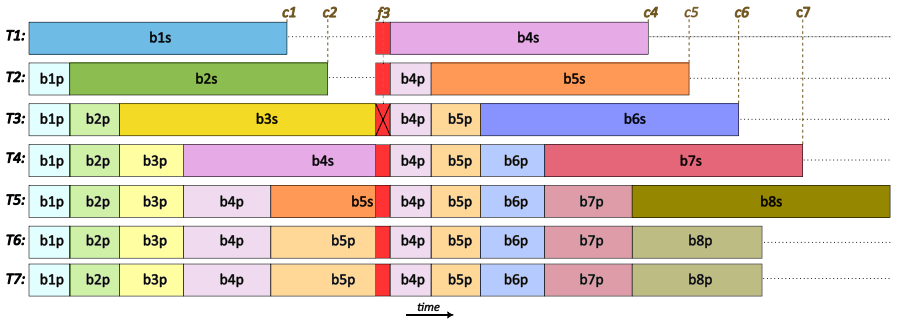



**Fig. 2** The first eight blocks of a speculative execution with seven threads where speculation fails in the third block

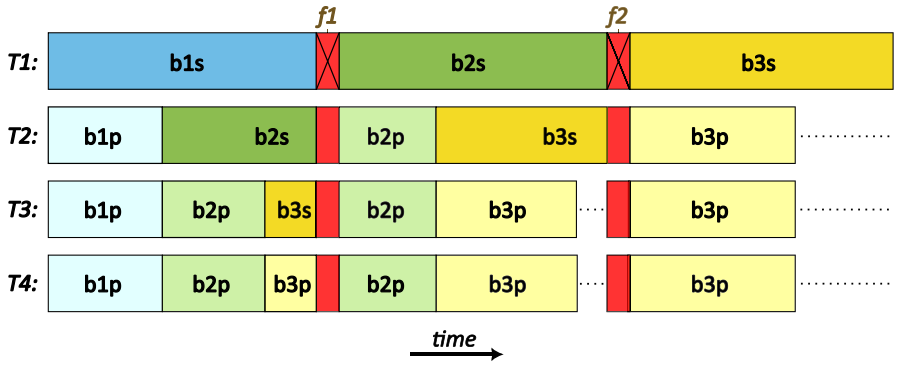



**Fig. 3** The first three blocks of a speculative execution with four threads where all speculative executions fail

Although it is not appreciated in the figures due to the small number of blocks used, as long as continuous successful validations occur, our model tends to a balanced state where sequential and parallel blocks are interleaved in such a way that approximately half of the threads are in charge of parallel versions and the other half of sequential versions. As soon as a failed validation occurs, the recovery process resets the state to a configuration similar to the initial one. With this scheme, the situation where there are almost no threads available to schedule the parallel execution of a block is unlikely, and it generally only happens when a very low number of threads is used. In this situation, the model will schedule the parallel version of the block for a number of threads greater than the currently available, and the associated sub-blocks will wait to be executed as soon as threads becomes available.

According to the process described, the highest possible speedup that can be obtained is $p/2$, where $p$ is the number total of threads, since the program must be executed in duplicate, the sequential and the parallel version. Other TLS models have the potential to achieve higher theoretical speedups, although in practice, they are greatly limited by the high computational cost necessary to detect and manage data dependencies. This degrades the performance significantly, being in many cases lower than that obtained with the code duplication method, as we will discuss in Sect. 5. In any case, the limit above is an ideal theoretical maximum that does not take into account some factors, as several intrinsic design waitings have to be taken into account:

- When the execution of the sequential version of a block ends, that thread is free, available and waiting for new work. However, it will not be until the planning of the next chunk that it will be possible to give work to this thread. This scheduling of the next chunk will occur when the execution of the current speculative parallel block ends. Therefore, this waiting time does not result in useful work. These waits depend on the program's flow and vary depending on the code being parallelized. However, they will be less relevant when the number of total threads grows since the execution time of the parallel versions for each block is

significantly reduced. Therefore, the wait between the time a thread is freed and the time it starts the execution of the next block is shorter.

- The model bases its performance on the anticipated advancement of work. Note that in the execution of the last blocks, no more work can be anticipated, and it will be necessary to wait for the completion of the sequential version of these last blocks to obtain the final correct results. As the number of blocks in the model increases, this effect is reduced, since only the last blocks are affected, and these will be smaller.
- In Figs. 1, 2 and 3 shown above, the time required to validate the results has not been represented, since usually the time used by these checks is negligible. However, for large or complex data structures, the checks can take significant time that should be considered.
- Finally, the time lost in failed parallel speculations and in the execution of blocks that start from incorrect speculative values has to be considered. After a failed verification, all executions of subsequent blocks must be cancelled and the execution must be restarted with the next block with correct verified values. A failed speculative parallelization of a block means that all the time advanced by this parallelization is lost, in addition to any work done by the following blocks, since they started from non-valid speculative values and their own validation was dependent on the validation of the previous blocks. The impact on the final performance is greatly affected by these failures, as it results in much of the work done being invalidated and repeated.

In general, we can conclude that, at the design level, the model presents fewer idle times and it approaches the maximum theoretical speedup of $p/2$ as the number of threads and iteration blocks increases, resulting in smaller blocks. The performance will be closer to optimum with a greater number of correct speculations. Usually, the number of successes in the speculation increases the smaller the block size is, since a data or calculation conflict is less likely to occur. However, this does not always happen and it also depends on the parallelized code.

Choosing a small block size improves the distribution of work among threads for the parallel execution version. However, there are also other factors of the model design that significantly affect performance. The main one is the overhead present in the implementation used, which includes, but is not limited to, synchronizing threads, scheduling the launch of each block of iterations, and making copies of the data so that there are two versions of the data for each block, one for the sequential run and the other for the parallel run. This overhead becomes more significant the smaller the grain of each block is in relation to the internal overheads of the implementation.

It is also interesting to analyse the additional memory usage required by the proposed model, since it is based on duplicate code execution and nesting of speculations, both techniques requiring multiple versions of the same data. A copy of the speculative variables is needed for each version of each block. These data must be kept in memory until the block is validated. To validate a block, all the previous ones must also be validated. Therefore, at a given time, the number of copies needed is related to the nesting level of the blocks, being this the number of blocks that have

been executed but not validated. This value should be multiplied by two to obtain the total number of copies, since two versions of the data exist for each block, one for its sequential version and another for its parallel version.

Note that usually the speculative nesting depends only on the number of total threads used. As mentioned previously, over time and as long as failed validations do not occur, the system tends to a balanced state where half of the threads execute sequential blocks and the other half parallel blocks, resulting in a nesting value close to $p/2$. At the beginning of the execution or after a failure is when the highest levels of nesting are reached, with an initial theoretical recurrence limit of $p - 1$, although in practice, this limit is not reached in most cases unless a low number of threads is being used. As two copies per block are needed, the average memory footprint usage in each case is twice the indicated nesting value.

In summary, the memory footprint of the model is directly related to the number of threads used, reaching a maximum of $2(p - 1)$ copies of the speculative data shortly after the beginning of execution and after each failed validation, although in reality this maximum is somewhat lower when more than few threads are used. If no failures occur in the speculation, the nesting level is reduced over time until reaching an average memory usage of roughly $p$.

## 4 SpecLib: an automatic speculative parallelization library

In this section, the new proposed library, called `SpecLib`, will be detailed. It implements the model described in Sect. 3. We will begin with its syntax and semantics and continue with the implementation and optimization details.

### 4.1 Syntax and semantics

```
SpecLib::Configuration cfg{NThreads, MinParalThreads};
SpecLib::specRun(cfg, begin, end, step, iters_chunk, f, args);
```

Listing 1: Call to the main function of the SpecLib library.

The main objective of the library is to parallelize `for` loops speculatively. For this, users must call the library's `specRun` function, providing the necessary information shown in Listing 1. The required parameters are described below.

Three of the input data needed are those that define a conventional `for` loop, i.e., the *begin* value of the loop, its *step*, and the *end* value, which when reached by the index of the loop indicates its completion. The library supports positive and negative values and different integer data types for all of them. Following the model specifications, the library internally divides the total number of iterations into different chunks or blocks of iterations.

The user specifies the number of iterations of these chunks as the parameter *iters_chunk*. This is an essential parameter for library performance, and a balance must be struck between being small enough to reduce certain waits and increasing the number of successful speculations, while being large enough so that library overhead does not impact performance too much. The helper function `getChunkSize(totalIterations, numChunks)` can be used to compute the chunk size when the user prefers to provide a desired number of chunks. This optimal value will significantly vary depending on the type of parallelized code, the execution environment and many other factors. For this reason, the library allows the user to set this parameter.

The `Configuration` object of the library allows to set various configuration parameters for the parallel execution. The first is the number of threads to use, three being the minimum required. Typically, this value is set to the number of cores available for execution. The higher the number of threads used, the more threads the speculative version can run in parallel, usually meaning higher performance. Scalability also depends on other factors, including the code to be parallelized, thus sometimes increasing the number of threads may not improve performance. The second parameter of this configuration object, `MinParalThreads`, is optional. The speculative parallel version of a block will be distributed among the maximum between the number of available threads during its scheduling and this parameter value. Its default value is 2, which is the minimum supported. The default value is almost always the best option, but in some niche cases, especially for a low number of threads where the situation of there being very few threads available is more frequent, increasing this value can slightly improve performance, so users can configure it for fine-tuning, although it is usually unnecessary.

Finally, the library `specRun` function receives as a parameter the function `f` that contains the body of the loop to be speculatively parallelized, and a variable list of parameters `args` that are the speculative variables. This function can be statically defined as a regular C++ function, a function pointer, or a C++ lambda function. In this function, the speculative variables are those whose result could be affected by parallelization. A chunk's sequential and parallel versions receive different instances of these variables, which will be compared when both executions finish. If the values of the two versions match, the speculative execution will be regarded as valid, while its results will be discarded otherwise.

The function `f` must receive the current iteration of the loop as its first parameter, and the speculative variables as its following parameters. This function will be called by the library in each iteration, so that its body corresponds to the body of the loop. `SpecLib` supports an alternative version of `specRun` that allows to define the function in a more customized way, so that instead of being called in each iteration, it is only called once per chunk. Of course, this implies receiving additional information, such as the limits of the chunk and its step, as well as performing the iterations of the chunk within the function. While this alternative requires more effort from the users, it can reduce the overhead associated with the invocation of the user-defined function, and it provides more control over the execution of the loop as well as access to some useful information about the current execution state. In addition, we observed that some compilers sometimes find optimizing the loops

defined in this function easier than those used in the basic form, especially if the compiler fails to inline this function for some reason or limitation.

```
#include "speclib/speclib.h"
#include <iostream>

int main() {
  unsigned int Vals[10000]; // declare array of 10000 elements
  set_array(Vals); // calls a function that sets the values of the array

  // lambda function to pass to the library with only one speculative
       variable (result) and capturing by reference a shared variable (Vals)
  const auto func = [&Vals](const size_t iter, unsigned int& result) {
    if (Vals[iter] > result) {
      result = Vals[iter];
    }
  };
  // get the number of iters of each chunk for a total of 40 chunks
  const size_t chunkSize = SpecLib::getChunkSize(10000, 40);

  unsigned int res = 0; // speculative variable result

  // call to SpecLib main function (8 threads, iters from 0 to 10000 with
       step 1)
  SpecLib::specRun({8}, 0, 10000, 1, chunkSize, func, res);

  // print results
  std::cout << "Maximum value of array: " << res << std::endl;
}
```

Listing 2: Speculative parallelization on a loop to obtain the largest value of an array of 10000 elements using 8 threads using `SpecLib`.

Listing 2 illustrates the usage of the library in a code that searches for the largest value in an array. The call to the `specRun` library function requests the use of 8 threads for a loop of 10000 iterations, starting at 0 and ending at 10000 with step 1. The user employs the helper `getChunkSize` function to compute the number of iterations per chunk to use a total of 40 chunks. The last two parameters are the lambda function `func` in which the main work is done and a single speculative variable `res` where the result will be obtained.

The library supports the definition of different compilation flags to expand its functionality or modify its behaviour. We highlight the possibility of extracting statistics on the number of successful and failed speculations, which is useful when analysing the library performance. It is also possible to extract detailed information about the execution times to know how much time is spent in each part of the parallelization. Another compilation flag allows the simulation of a certain percentage of correct speculations. The results obtained when the library is used in simulation mode will be incorrect. Still, its utility lies in figuring out the performance that can be obtained from the parallelized code for a given success rate. Finally, one last option can be used to avoid the check that the library performs in each iteration

of the loop to cancel the execution of the current chunk if it is discovered that it is being run on variables obtained from failed speculations. This check is convenient in most cases, as it allows to prematurely terminate the execution of chunks whose work will not be useful. However, although it is not common, sometimes this check can penalize performance since it can prevent the compiler from applying some optimizations.

#### 4.1.1 Helper wrapper classes

The default behaviour of the library with speculative variables may not always be adequate or optimal in all circumstances. That is why the library provides a set of easy-to-use and highly efficient helper wrapper classes for the most common problems.

Operations on floating-point numbers are present in many applications nowadays, and they are characterized by the presence of rounding and precision errors in their operations, which accumulate after successive computations. Therefore, certain operations that mathematically satisfy the associative property and should not affect the result by changing the order in which they are executed can provide different results when performed on floating-point numbers due to these issues. The C++ comparison operation is used by the library in the validation process of a speculative variable, and for floating-point types identical values are required to satisfy equality. This may be the desired behaviour sometimes, but since the parallelization of operations often changes their order of execution, what is often desirable in this situation is to allow a certain range of tolerance for the comparison.

For this reason, our library provides several wrapper classes for floating-point types that allow setting a tolerance level for comparisons. This is not a universal solution that can be applied to any problem. Floating-point precision errors have numerous repercussions, so how they affect the program and the possible workarounds must be studied in each case. The implications could be profound, and a complete rethinking of the algorithm may be necessary. Even so, one of the solutions typically used is to simply accept the existence of a certain margin of error in the results due to these imprecisions. In these cases, the floating-point helper classes are also helpful, although their primary purpose focuses on the precision errors caused by the associativity of operations. These classes overload all the operations supported by the native types, so that they can be used directly in the same way and thus their use requires very little code modification. It is in the overloaded comparison operators where the margin of error is taken into account.

Different types of tolerance comparisons are supported. The simplest comparison type is an absolute numerical comparison where if the absolute value of the difference between the two numbers to be compared is less than the tolerance value, these numbers will be considered equal. A second type is to use a relative tolerance comparison. The idea of a relative comparison is to find the difference between the two numbers and to consider how large it is compared to their magnitudes. The space between two numbers representable in the floating-point form is known as the unit of least precision or ULP. It is commonly used to measure the precision of the type used, and the difference in ULPs between two floating-point numbers can also be

used in comparisons. One issue of the relative and ULP comparisons happens when comparing numbers close to zero. In these cases, two near-zero numbers that are close to each other can be considered different. In this circumstance, a hybrid absolute and relative or hybrid absolute and ULP comparison will be the best approach, where the absolute tolerance value is used first to check whether the numbers are really very close, being in that case already considered equal. If not, then the relative or ULP comparison will determine their equality. All these options allow for very custom and defined use of tolerant comparisons for floating-point types as needed.

Arrays, vectors, lists, and other standard C++ containers are commonly used and allow the user to implement data structures containing sets of values easily. These types can be used directly in the library in a speculative way, but the user must be aware of the implications of their use, especially if the data containers are huge, as they could affect performance. In the context of speculative container variables, the equality check means checking that the number of elements in them is the same and that all their elements are equal, as only in this case the two containers are considered equal. The library internally needs to continuously copy the value of the speculative variables since for each new chunk, the value of the speculative variables computed by the previous chunk must be copied, and then these must be copied again so that a different copy is available for sequential execution and another for parallel execution. This is the expected behaviour, but in many situations, all these copies are unnecessary and can be optimized.

A specific case of optimization would be when the user knows that in a given iteration `i` the collection `col` element is always accessed with the same offset `col[i]`, something that is quite common. For this case, the library includes a helper wrapper class, with which the user can directly encapsulate a set of consecutive values in memory, be it an array, a vector, or even a pointer, and access its elements in the usual way. When the library prepares a new chunk, and this class is used, instead of making a complete copy of the array, only the elements that are going to be accessed in those iterations of the loop will be copied, thus avoiding copying large amounts of data. At the time of performing the speculation check, only these values will be checked. This reduces memory usage and the amount of data on which to perform checks, which improves performance, especially if these checks involve classes with complex comparisons and not native types. However, in most cases, the great performance improvement comes from reducing the number of copy operations in memory. Unfortunately, in many codes, modifications on arrays happen with patterns that cannot be easily described, for example, because they use indirections or complex indexing functions. For this case, the library provides another similar wrapper class with the same purpose, but which supports total flexibility in the indexing of the array it represents. This is achieved at the cost of much higher storage and management overheads and a more challenging use of the class.

The library has a helper class that facilitates the parallelization of reduction operations defined by the user. The reduction operations imply reducing a diverse number of elements to a final result by applying the same operation continuously. They are associative and are usually easily parallelizable, but not directly, so the helper class follows a specific process that takes into account the independence of the individual operations instead of the temporally partially computed values. With

a direct use of the library without this class the reduction would produce an enormously speculation failure rates for the reduction variables if direct parallelization is attempted. The use of this helper class allows the reduction to succeed in almost all cases if it meets the appropriate properties, but requires a certain effort from the user to implement it.

In parallel algorithms, atomic variables are sometimes used to allow several threads to access the same variable atomically, guaranteeing that the read and/or write operation done on this variable is performed completely by one thread before another can perform any other operation on it. This type of operations are implemented and supported at the hardware level in most current processors for the basic integer types. Since C++11, the `std::atomic<T>` type has been introduced into the language, making it easy to define and use atomic variables and operations. However, this class presents specific characteristics and limitations that make it initially incompatible with the library if intended to be used as a speculative variable. With the helper wrapper class that the library provides, the atomic types can be used as result speculative variables since it internally implements an atomic variable `std::atomic` but in a way that is compatible with the library and expanding the type with more functionalities.

### 4.2 Implementation and optimizations

Our proposal is implemented as a header-only C++ library designed and developed to allow a high performance level and flexible ease of use. To achieve this, it internally makes intensive use of templates and multithreading techniques. Some applications may need to perform multiple calls to the library, so it is also relevant to achieve good performance in the library call itself, and not just in the internal execution. Although it largely depends on the operating system, creating and destroying threads is usually relatively slow. The library uses and manages a thread pool to avoid creating and destroying threads with each call, so new threads will be created when necessary. Still, after their usage, they will be kept inactive and ready to be reused in possible future calls to the library instead of being destroyed.

The library internally implements a lock-free strategy using several variables and atomic operations for synchronization. This allows to keep threads as active as possible and obtain a very good performance and scalability by avoiding expensive blocking operations. A single thread is responsible for scheduling the execution of each next block of iterations. The rest of the threads simply alternate between executing work or waiting for this thread to release new work. The validation of a block will be performed when all the three following points are reached for its execution: the execution of the sequential version ends, the execution of the parallel version ends, and the previous block is validated. Therefore, the last point to be reached is the one that triggers the validation. In the case of the first block and, in general, any block that starts from non-speculative verified values, it is not necessary to wait for a previous block to be validated, and the verification is carried out as soon as both the parallel and the sequential versions of the block finish. This flexibility in the validation process avoids unnecessary synchronizations and provides the necessary agility

to the process so that in the case of an erroneous validation, it can start recovery as soon as possible.

After a successful validation, the program does not need to perform any special action and can continue its normal execution. However, if a validation is unsuccessful, all currently running blocks must be stopped since they all started from wrong values. To stop the execution, the library modifies the value of a flag variable that its runtime usually checks in every iteration to detect this situation and terminate these blocks as soon as possible. Users can also control when to perform this check.

Finally, the optimization level that the compiler can apply is also very important. The heavily template-based library code is designed so that the compiler can optimize all its operations as much as possible. In particular, it must be capable of detecting and integrating the code provided by the user within the main loops of the library so that it applies the optimizations that compilers usually perform on loops. Although compilers are very advanced nowadays, with exceptional optimization capabilities, they tend to be considerably conservative sometimes. As a result, they do not always extract the necessary information from code or correctly apply some optimization techniques when the encapsulated code is provided in a user-defined function that is passed as an argument to functions and classes. Also, on other occasions, certain optimization parameter limitations prevent some techniques from being applied, although, in this case, these parameters are usually configurable and can be modified. In our tests with the most widely used modern compilers today, we have found that they almost always achieve a good level of optimization, although there are some exceptions.

## 5 Evaluation

In this section, the performance obtained by the `SpecLib` library is evaluated, and the results obtained are compared with one of the main alternatives, the `ATLaS` project [1], which is described in Sect. 2.

The benchmarks used and their sequential runtimes, which are taken as reference for the computation of the speedups of the speculative parallel versions, are described in Table 1. These times were obtained by running an optimized sequential version of each benchmark on a single core of the target system used in all the experiments, described in Table 2. All the parallel and sequential codes were

**Table 1** Benchmarks used

| Name | Problem Size (total iterations) | Seq. Time |
|---|---|---|
| B-synthetic | 30,000 | 22.714 s |
| R-test_reduction | 8,000,000,000 | 11.463 s |
| D-2D-Disc-Hull | 9,999,997 | 0.611 s |
| E-2D-Square-Hull | 9,999,997 | 0.719 s |
| F-2D-Kuzmin-Hull | 9,999,997 | 0.553 s |

**Table 2** System configuration

| Feature | Value |
|---|---|
| CPUs per Node | 2 × Intel Xeon Platinum 8352Y |
| CPU Family | Ice Lake |
| CPU Frequency | 2.20 GHz (Turbo: 3.40GHz) |
| Num Cores/CPU | 32 |
| Total Cores per Node | 2 × 32 = 64 |
| Memory per Node | 256GB DDR4 |
| Operating System | Rocky Linux 8.4 |
| Compiler | g++ 10.1.0 |

compiled using the same software environment, an optimization level `O3`, and every time measurement is the average of ten runs.

*B-synthetic* is a synthetic benchmark designed to produce many correct speculations, as only two iterations out of 30,000 are strongly likely to cause dependency violations. Even so, these two iterations have a great impact on the calculations made in the iterations after them since they modify the values of a small array that is used intensively to make calculations on a couple of reduction operations. As a result, their impact can be very large if the speculative process does not maintain good isolation of all speculative variables and does not implement a correct recovery system after speculative failures.

The synthetic benchmark *R-test_reduction* implements a simple addition reduction operation for a double-precision floating-point variable. This benchmark allows testing the performance of some wrapper classes explained in Sect. 4.1.1, namely the one designed to deal with variables involved in reductions and the one that allows establishing a tolerance for the validations of floating-point speculative variables. An alternative implementation using the wrapper class for atomic variables and a user-defined function that executes a chunk of iterations instead of a single one will also be tested, also using the floating-point helper class in this case.

The 2D-Hull problem solves the computation of the convex hull (smallest enclosing polygon) of a set of points in the plane. The probability of a dependency violation and the execution flow in the 2D-Hull algorithm depends on the shape of the input set and the size of the inner loops, which varies and can cause a certain imbalance. Internally the code speculates on an array, which also differs in size depending on the input data, where access through indirections is frequent, making the input even more decisive in the program's behaviour. Therefore, three different input sets with tens of millions of points each are used, and each one will be considered as an independent benchmark since as already mentioned, their behaviour, results and performance are very different. The Kuzmin input set *F-Kuzmin-2D-Hull* follows a Gauss-Kuzmin distribution, with a higher density of points around the center of the distribution space, leading to very few dependency violations, as points far from the center are very rare. The other two input sets, the Disc *D-Disc-2D-Hull* and the Square *E-Square-2D-Hull*, lead to more dependency violations than Kuzmin because their points are evenly

distributed within a Disc and a Square, respectively. The Square input set results in an enclosing polygon with fewer edges than the Disc input set, thus producing fewer dependency violations.

Figure 4 illustrates the impact of the chunk size in the final performance of the `SpecLib` benchmarks, using 48 threads in this example, i.e. one per core. Due to the different nature of the benchmarks, it is more convenient for this comparison to show the performance relationship concerning the total number of chunks instead of the size of each chunk. Note that the performance with respect to the chunk size is usually a concave downward curve because of the problems mentioned in Sect. 4.1 when the chunks are too large or too small. Interestingly, while some benchmarks present a limited number of chunk sizes that provide good performance, others can reach reasonable performance results for a large range of values.

In the experiments with the `SpecLib` versions, executions were performed varying the size of the block of iterations in order to choose the one that offers the best results. In the `ATLaS` versions, in addition to the size of the block of iterations, two additional parameters also affect performance that had to be considered. As a result, for these versions, multiple executions were also performed with different combinations of the three parameters to select the one that offers the best performance. It was also always checked that the outcome was correct for every test. The final performance results, which use between 3 and 64 threads, are shown in Fig. 5.

*B-synthetic* has three speculative variables, two variables on which a simple reduction operation is performed and a small array accessed via indirections. `SpecLib` effectively handles reduction operations with the reduction helper class, and the array can be treated directly. In this way, `SpecLib` maintains an optimal speculative success rate, where only two of the iteration chunks result in

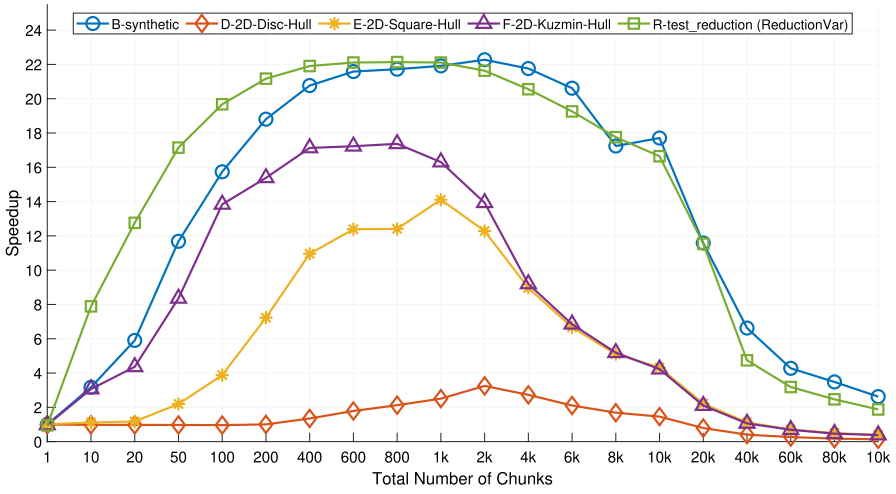
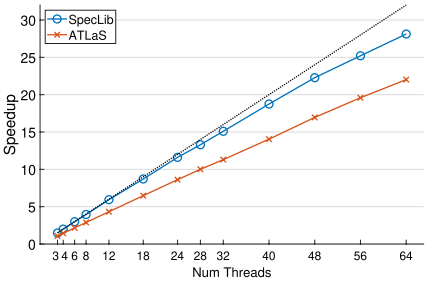
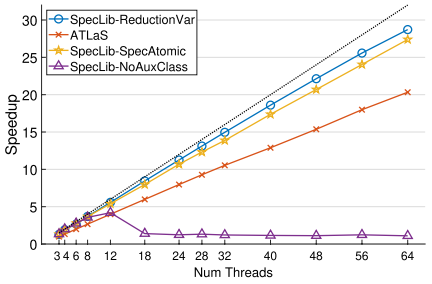


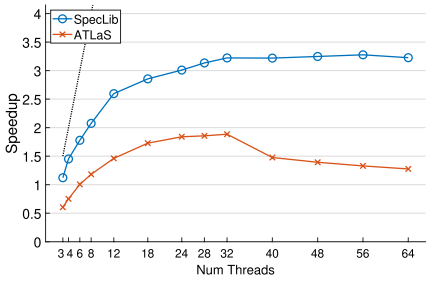

**Fig. 4** Performance of the benchmarks in a 48-core system using the `SpecLib` library varying the total number of chunks used



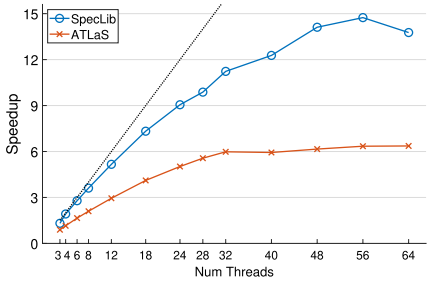

(a) *B-synthetic* benchmark



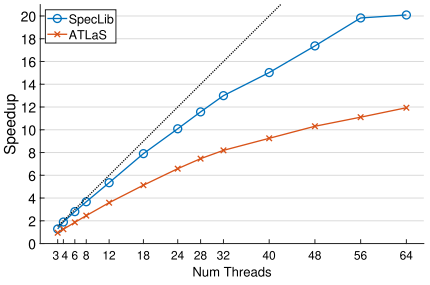

(b) *R-test_reduction* benchmark




(c) *D-2D-Disc-Hull* benchmark




(d) *E-2D-Square-Hull* benchmark




(e) *F-2D-Kuzmin-Hull* benchmark

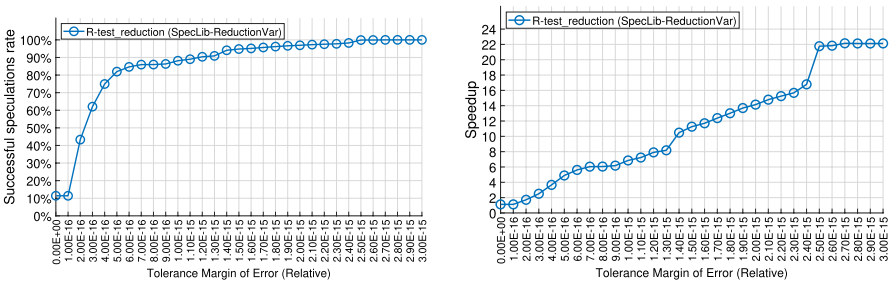
**Fig. 5** Speedups of the speculative parallel implementations of `SpecLib` and `ATLaS` (the dashed black line indicates the maximum theoretical speedup of the model on which `SpecLib` is based)

erroneous speculations in most cases. It, therefore, achieves optimal performance and efficiency even for a large number of cores, with an efficiency of 99.5% when using 4 threads and dropping only to 88% for a large number of 64 threads. `ATLaS` also performs very well on this test, albeit slightly worse.

In the *R-test_reduction* benchmark, the `SpecLib` library uses the reduction helper class to optimize its reduction operation. It also makes use of the helper wrapper class that allows setting a tolerance margin of error in the comparisons used in the validations, with a small but sufficient value to ensure that validations are both correct and successful most of the times. Because the value of the speculative variable reaches different orders of magnitude during execution, a relative

tolerance value is used. Since it is a reduction operation, speculative failures will only come from precision errors in floating-point operations. It is therefore interesting to note how the successful speculation rate and, thus, the performance are affected as the tolerance margin of error of the double-precision floating-point comparisons is altered, as we do in Fig. 6 for the 48-cores case. In this example, the success speculative rate is about 11% if a very low or no tolerance is used, resulting in a 1.1x speedup. As an increasingly higher tolerance value is established, the number of valid speculations rises, reaching a 100% rate and a speedup of 22.1x with a relative tolerance value of 2.70E-15 or higher. This experiment also illustrates that the speculative success rate significantly affects the final performance nonlinearly. While the success rate grows very quickly as the margin of error used increases, performance grows more slowly and progressively.

For the `SpecLib-ReductionVar` case, with the use of the reduction helper wrapper, it can be ensured that the performance and scalability of this benchmark is close to the optimum, with high efficiency levels between 86% and 95%. While this is the most efficient implementation with `SpecLib`, comparing it with other alternative library implementations is interesting, as shown in Fig. 5b. The `SpecLib-SpecAtomic` version uses instead the atomic wrapper class introduced in Sect. 4.1.1 to allow the use of atomic operations on this floating-point variable. As atomic operations are expensive, their continued use would produce poor performance. For this reason, this benchmark variation uses a custom loop in which a standard local variable is used within the loop instead of the atomic speculative one, thus performing only a single speculative atomic operation at the end of this custom loop, considerably improving the performance. The `SpecLib-NoAux-Class` variant does not use any of these auxiliary classes. Instead, it directly parallelizes the original loop, obtaining very poor performance, proving that these auxiliary classes are necessary for these types of operations. These benchmarks show that the reduction helper class performs its function in a very optimal way. Let us notice that the atomic wrapper class allows the use of atomic operations within the speculative loop, which can be useful for certain specialized uses. `ATLaS` also implements specialized methods for dealing with reduction




(a) Speculative success rate results (b) Speedup results

**Fig. 6** Speedup and rate of successful speculative block validations for the `SpecLib` *R-test_reduction* benchmark using 48 cores and varying the tolerance of the double-precision floating-point comparisons

operations, thanks to which it achieves a decent performance. However, it is significantly lower than the one obtained by `SpecLib`.

The problems inherent to the algorithm of the 2D-Hull benchmarks affect the scalability and the maximum performance obtained. It should also be noted that the sequential time of these benchmarks is very low, less than one second, as shown in Table 1. Parallelizing benchmarks with such a low sequential execution time considerably penalizes scalability, as it is more difficult to obtain an adequate block size with low overhead and a success speculative rate for a high number of threads, and the time impact of these speculative failures is more significant. Despite all these issues, the results achieved for these benchmarks are decent, given their particular circumstances. The *F-Kuzmin-2D-Hull* benchmark achieves a maximum speedup of 20.1× with 64 threads, although it hardly shows any improvement from 56 threads due to these scalability issues, for which a 19.8× speedup is obtained. This also happens for the *E-Square-2D-Hull* benchmark, where performance is even penalized when using 64 threads, achieving a maximum speedup of 14.7× with 56 threads. The *D-Disc-2D-Hull* benchmark has many dependency violations. This benchmark begins to suffer in its scalability from 18 threads, for which a speedup of 2.9× is obtained. The maximum speedup achieved is 3.3× for 56 threads, although this is a very minor improvement compared to the 3.2× and 3.1× speedup achieved for 32 threads and 28 threads, respectively. `ATLaS` shows considerably lower performance in these benchmarks, presenting more pronounced scalability problems.

## 6 Conclusions

Many loops present characteristics that preclude their automatic parallelization and even a traditional manual parallelization using standard tools such as OpenMP. This is the case of loops in which we are unsure about dependencies among different iterations. In these situations, only a speculative approach, where the results of the parallelization are verified against a correct execution to make sure that the results are valid and take corrective actions otherwise, enables the exploitation of parallelism in these codes.

In this paper, we present a new thread-level speculative parallelization model and its implementation in a high-performance library named `SpecLib`, designed to automatically and efficiently parallelize loops that are not amenable to other approaches, for example because of indirections, race conditions, or data dependencies in the original code. Our library makes transparent to users the great internal complexity of distributing, synchronizing, verifying, correcting and restarting iterations when necessary using efficient shared memory mechanisms.

Particular emphasis has been placed on keeping the library as efficient as possible to achieve competitive performance. For this, certain flexibility has been granted to the library's configuration, which allows users to adjust the size of the block of iterations, the most crucial parameter whose optimal value varies according to the parallelized loop. At the same time, the library presents other advanced options for experienced users that enable finer custom tuning and the use of more complex techniques. The set of helper wrapper classes included with the library must also be

highlighted, as it dramatically expands its possibilities and improves performance. It is especially useful for handling reductions, floating-point numbers, and containers such as arrays, among others.

The library achieved competitive performance in the different benchmarks used in the evaluation. While the scalability is very good for benchmarks where the number of correct speculations is high, it is somewhat limited for the difficult cases with many failed speculations. This reveals the great cost of discarding the work advanced by the multiple nested speculations.

As future work, we propose the creation of new helper classes to cover a greater diversity of cases and improve performance, especially those aimed to optimize access to arrays or other commonly used data structures in ways not considered in this paper. Also, it would be interesting to transfer the ideas presented in this paper to a distributed memory environment to increase scalability if we found instances where this could be useful. However, the challenges are immense and the overhead and communications costs associated with this task are huge, so for each proposal it would also be necessary to explore its feasibility. Finally, we also consider it interesting for future work to estimate the energy consumption of our proposal and present modifications that minimize it if possible.

**Funding** Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This research was supported by Grants PID2019-104184RB-I00, PID2019-104834GB-I00, PID2022-141623NB-I00, and PID2022-136435NB-I00, funded by MCIN/AEI/ 10.13039/501100011033, PID2022 also funded by "ERDF A way of making Europe", EU, and the predoctoral Grant of Millán Álvarez Ref. BES-2017-081320, and by the Xunta de Galicia co-founded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups (ED431C 2021/30 and ED431C 2022/16). Funding for open access charge: Universidade da Coruña/CISUG. We also acknowledge the support from the Centro Singular de Investigación de Galicia "CITIC" and the Centro Singular de Investigación en Tecnoloxías Intelixentes "CiTIUS", funded by Xunta de Galicia and the European Union (European Regional Development Fund- Galicia 2014-2020 Program), by grants ED431G 2019/01 and ED431G 2019/04. We also acknowledge the Centro de Supercomputación de Galicia (CESGA).

**Data availability** The library implementation, some code examples, and the benchmarks used will be made publicly available upon acceptance of this manuscript at https://github.com/UDC-GAC/speclib.

## Declarations

**Ethical approval** Not applicable.

## References


1. Aldea S, Estebanez A, Llanos DR, Gonzalez-Escribano A (2016) An OpenMP extension that supports thread-level speculation. IEEE Trans Parallel Distrib Syst 27(1):78–91
2. Bhowmik A, Franklin M (2002) A general compiler framework for speculative multithreading. In *Proceedings of the Fourteenth Annual ACM Symp. on Parallel Algorithms and Architectures*, SPAA '02, pages 99–108, New York, NY, USA. ACM
3. Cintra M, Llanos DR (2005) Design space exploration of a software speculative parallelization scheme. IEEE Trans Parallel Distrib Syst 16(6):562–576
4. Dang FH, Yu H, Rauchwerger L (2002) The R-LRPD test: Speculative parallelization of partially parallel loops. In: *Proceedings 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, p. 318, USA. IEEE Computer Society
5. Kelsey K, Bai T, Ding C, Zhang C (2009) Fast track: A software system for speculative program optimization. In *2009 International Symposium on Code Generation and Optimization*, pp. 157–168
6. Kulkarni M, Burtscher M, Cascaval C, Pingali K (2009) Lonestar: a suite of parallel irregular programs. Commun ACM 52(9):65–76
7. Kulkarni M, Pingali K, Walter B, Ramanarayanan G, Bala K, Chew LP (2007) Optimistic parallelism requires abstractions. *SIGPLAN Not.*, 42(6):211–222
8. Kumar S, Singh SK, Aggarwal N, Aggarwal K (2021) Evaluation of automatic parallelization algorithms to minimize speculative parallelism overheads: an experiment. J Discrete Math Sci Cryptography 24(5):1517–1528
9. Kumar S, Singh SK, Aggarwal N, Gupta BB, Alhalabi W, Band SS (2022) An efficient hardware supported and parallelization architecture for intelligent systems to overcome speculative overheads. Int J Intell Syst 37(12):11764–11790
10. Matsunaga D, Nunome A, Hirata H (2019) Shelving a code block for thread-level speculation. In *20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 427–434
11. Ottoni G, Rangan R, Stoler A, August DI (2005) Automatic thread extraction with decoupled software pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 12 pp.–118
12. Quiñones CG, Madriles C, Sánchez J, Marcuello P, González A, Tullsen DM (2005) Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. SIGPLAN Not. 40(6):269–279
13. Rajwar R, Goodman JR (2001) Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pp. 294–305, USA. IEEE Computer Society
14. Rauchwerger L, Padua D (1995) The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. SIGPLAN Not. 30(6):218–232
15. Salamanca J, Baldassin A (2019) A proposal for supporting speculation in the OpenMP taskloop construct. In *OpenMP: Conquering the Full Hardware Spectrum*, pp. 246–261, Cham. Springer Intl. Publishing
16. Salamanca J, Baldassin A (2023) Evaluating the performance of speculative doacross loop parallelization with taskloop. *arXiv preprint* arXiv:2302.05506
17. Tian C, Feng M, Nagarajan V, Gupta R (2008) Copy or discard execution model for speculative parallelization on multicores. In *2008 41st IEEE/ACM Intl. Symp. on Microarchitecture*, pp. 330–341
18. Tian C, Feng M, Nagarajan V, Gupta R (2009) Speculative parallelization of sequential loops on multicores. Int J Parallel Program 37(5):508–535
19. Torrellas J (2011) Speculation, thread-level. In *Encyclopedia of Parallel Computing*, pp. 1894–1900. Springer, Boston
20. Tournavitis G, Wang Z, Franke B, O'Boyle MFP (2009) Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. SIGPLAN Not. 44(6):177–187
21. Xekalakis P, Ioannou N, Cintra M (2009) Combining thread level speculation helper threads and runahead execution. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pp. 410–420, New York, ACM

22. Zilles C, Sohi G (2002) Master/Slave speculative parallelization. In *Proceedings of the 35th Annual ACM/IEEE Intl. Symp. on Microarchitecture*, MICRO 35, pp. 85–96, Washington. IEEE Computer Society Press