Graphical Abstract

ClangOz: Parallel Constant Evaluation of C++ Map and Reduce Operations

Paul Keir, Andrew Gozillon

Highlights

ClangOz: Parallel Constant Evaluation of C++ Map and Reduce Operations

Paul Keir, Andrew Gozillon

- Research highlight 1
- Research highlight 2

ClangOz: Parallel Constant Evaluation of C++ Map and Reduce Operations

Paul Keir[a,*], Andrew Gozillon[b]

[a]*School of Computing, Engineering and Physical Sciences, University of the West of Scotland, High Street, Paisley, PA1 2BE, United Kingdom*
[b]*Advanced Micro Devices AB, Nordenskioldsgatan 11 A, Office 233, 211 19, Malmö, Sweden*

Abstract

Interest in metaprogramming, reflection, and compile-time evaluation continues to inspire and foster innovation among the users and designers of the C++ programming language. Regretably, the impact on compile-times of such features can be significant; and outside of build systems, multi-core parallelism is unable to bring down compilation times of individual translation units. We present ClangOz, a novel Clang-based research compiler that addresses this issue by evaluating annotated constant expressions in parallel, thereby reducing compilation times. Prior benchmarks analysed parallel map operations, but were unable to consider reduction operations. Thus we also introduce parallel reduction functionality, alongside two additional benchmark programs.

Keywords: Compiler, Parallelism, C++, Partial Evaluation

1. Introduction

Compile-time metaprogramming in C++ has been of interest since the discovery that C++ templates were Turing-complete [1]. Exploration of compile-time metaprogramming has resulted in the addition of generalised constant expressions to the language; a concept proposed in 2003 [2]; and

*Corresponding Author
Email addresses: paul.keir@uws.ac.uk (Paul Keir), andrew.gozillon@amd.com (Andrew Gozillon)

added in C++11 [3] with the inclusion of the *constexpr* specifier. A constant expression is an expression which would remain constant at runtime and could thus be evaluated during compilation. The *constexpr* specifier allows functions and variable declarations to assert that they can be evaluated during compilation. With the addition of this specifier, compile-time programming has become more approachable, with a syntax almost identical to runtime code.

Since the addition of constant expressions to C++, the standard library specification has begun to incorporate support for both compile-time and runtime execution for its functionality. With the increasing *constexpr* support, larger program segments can now be constant evaluated. However, as more components are evaluated during compilation, so too do compile times increase. Adding parallelism to a program can help increase performance when used correctly. Yet parallelism is currently only available in runtime contexts; there is no existing concept of C++ compile-time parallelism.

We here introduce ClangOz [4], an experimental Clang-based [5] compiler which adds support for the parallel execution of for-loops within constant expressions. ClangOz seeks to give users control of parallelism through compiler intrinsics. These intrinsics are a set of functions built into the compiler, which may be utilised to convey information about the algorithm being *constexpr*-parallelised to ClangOz. A higher level application programming interface (API) is also provided which builds upon the *execution policy* overloads of existing standard C++ runtime library functions such as *std::for_each*, to allow easier access to *constexpr* parallelism.

In this substantially extended version of our recently published research papers [6, 7], we introduce a completely new foundation for our extended *constexpr* C++ standard library support: C'est 2, which is built upon the GNU C++ Library [8]. A consequence of this is that the C++ standard library namespace can be used throughout our updated benchmarks without conflict. On a similar vein, we now deprecate use of a non-standard custom *ce_par* execution policy object, in favour of the standard C++ library's *std::execution::par*. We also introduce two new benchmarks exploiting *reduction*; a word counting program, and a 3D heat solver. Both utilise different overloads of our fresh implementation of the C++ *std::transform_reduce* function template from the standard Numerics library; support for which is also newly presented here. Furthermore, with most transcendental mathematical functions within the C++ standard library unavailable during constant evaluation (we await implementations of P1383 [9] for C++26), we have

updated relevant benchmarks to make use of the GCE-Math library [10], and present a performance comparison of those benchmarks, in comparison to versions using our pre-existing ad-hoc *constexpr* maths routines; based on the GNU C Library (glibc) [11]. While measuring the granular durations of *constexpr* parallel regions requires custom functionality of the ClangOz compiler, we also now analyse the coarse black-box timings for full invocations of recent GCC and Clang compilers on our benchmark suite. Finally, we include a new section reviewing real-world Github repository uptake of C++ code aligned with the potential for *constexpr* parallelism.

A survey of related and relevant literature is presented in Section 2. Section 3 covers the ClangOz compiler, discussing its architecture; parallel constant expression evaluation; and intrinsics library, along with a concise implementation example of *std::for_each*. Most of our benchmarks utilise custom *constexpr* versions of C++ standard library components, and Section 4 discusses these aspects of our new C'est 2 library. Section 5 reports on experiments, with benchmarks facilitated by the novel compile-time parallelism feature, and considers performance and scaling in comparison with serial counterparts. Before we conclude in Section 7, Section 6 examines the broader uptake of the *constexpr* paradigm by C++ library authors using the Github website.

2. Background

Parallelism within compilers is not new, and much research has been undertaken, aiming to speed up different phases of the compiler. For example, investigation on the parallelisation of parsing [12], assembling [13], semantic analysis [14], lexical analysis [15] and code generation [16] compiler phases have been conducted. Despite this, most modern compilers avoid the additional complexity of adding parallelism for performance. The research presented in this paper differs from the prior art as it pertains to a smaller segment of the compiler; a subsection of the semantic analysis process. Nevertheless, it does adds an overhead for compiler developers; even if it is smaller in scope. This work is also distinct in placing the parallelism into the users' hands through an API, making the parallelism explicitly programmable.

The landscape of C++ compile-time programming has continued to expand in recent years. In the 2020 iteration of the language (C++20 [17]) provision for dynamic memory allocation and deallocation during compilation was adopted [18]; so allowing the creation of variable size containers

within constant expressions. C++20 also introduced a feature called *Concepts* [19], permitting a user to constrain template instantiation according to composable boolean predicates. Concepts allow for increased user defined type-safety within code bases, while improving error messages. Further proposals are pending, with the most interesting possible additions being metaclasses [20] and static reflection [21]. The former allows users to define a compile-time function that manipulates how a class's definition is generated; for example to make member functions of a class public by default. The latter allows deeper compile-time introspection of types; for instance, to check the names of class members.

Such developments in the language specification have allowed for projects that were previously impossible. The processing of regular expressions during compilation [22], static reflection through a library rather than the language [23], big-integer computation [24] and compile-time functional composition [25] are prominent examples. Such projects require a sizable amount of computation during compilation, and could benefit from acceleration by *constexpr* parallelisation.

Providing language features to allow processing during compilation is not unique to C++. Lisp [26], D [27], Rust [28], Julia [29], Elixir [30] and Circle [31] all give various compile-time facilities. Lisp was the first language with Turing-complete compile-time functionality; Lisp provides this feature in the form of macros which, unlike C-style macros, can perform computation as well as text substitution. The D programming language has many similarities to C++. Its compile-time features are based upon it, with the intention of simplifying them. The D language allows compile-time programming using constructs similar to C++ templates and constant expressions, though it extends these concepts with the introduction of eponymous and nested templates. In contrast: Rust, Julia and Elixir make use of Lisp-style macros that manipulate the AST for their compile-time metaprogramming. Rust and Julia also support compile-time computation through constant expressions that share similarities with C++. The Circle language is interesting as it builds on-top of C++17, adding a host of new data-driven metaprogramming features including range operators and pack generators. The concepts introduced in this paper could in turn be extended to such programming languages, having similar compile-time capabilities, albeit in a different guise when the capabilities are macro based.

The evaluation of constant expressions during compilation has parallels in the field of partial evaluation [32], where programs are specialised dynam-

ically at runtime, or statically during compilation, to achieve better performance. Partial evaluation of programs can lead to optimisations including constant folding; code simplification; strength reduction; and control flow optimisation. All such optimisations are possible through the explicit utilisation of the *constexpr* keyword within C++ to specialise code; and we recognise that the comparison of partial evaluation and C++'s compile-time features has been made before [33]. Research into partial evaluation and its applications have been ongoing for many years; and recently applied to the field of High-Performance Computing with the aim of increasing performance in a myriad of ways. For example, using static partial evaluation to optimise memory access patterns on the GPU [34]; creating domain specific languages utilising partial evaluation to facilitate high-performance libraries for accelerators [35]; and in the development of compilers and interpreters for dynamic languages that utilise partial evaluation to speculatively optimise code [36]. In a similar vein to the work in this paper for compile-time evaluation, some research on parallel evaluation of partial evaluation has also been conducted. Some examples are the parallelisation of a partial evaluator utilised in the specialisation of mutually recursive procedures [37]; distributed parallelisation of partial evaluators within programming languages [38]; and parallelisation of partial evaluations within evolutionary algorithms [39].

3. Compile-Time Parallelism

The ClangOz[1] compiler builds on Clang by adding parallelisation support to *for* loops in specific *constexpr* contexts. This is performed using an API of four intrinsic functions[2], used to communicate to the compiler how a loop should be parallelised. The intrinsic calls are placed within the function body containing each targeted loop; and assist with loop dependency analysis [40].

ClangOz uses these intrinsics to gather the information required to partition the loop body across multiple CPU threads. The intrinsic calls must be within a *constexpr* function; and adjacent to a corresponding *for* loop.

We also provide a higher-level API, wherein the parallelised loops and reductions, carefully annotated using the ClangOz low-level intrinsics API, are abstracted over by the high-level C++ standard Algorithms and Numerics libraries. Each of the many relevant function templates therein al-

[1]The compiler has no connection to Mozart and the Oz language.

[2]These are not *Clang* intrinsics per se; though they perform a similar role.

ready include a C++ execution policy class as their first parameter in a set of parallelising overloads introduced in C++17 [41]. An execution policy parameter allows an algorithm designer to overload a function's behaviour based on the *type* of each distinct policy. For example, a *std::for_each* function may be passed an execution policy object, directing the *std::for_each* to an overload that has a parallel implementation. For ClangOz, providing a *std::execution::parallel_policy* object, such as the globally declared *std::execution::par*, will execute the underlying low-level *loop* or *reduction* operation in parallel when evaluated within a constant expression.

Only C++ iterators targeting *contiguous* data are currently supported. This is not a fundamental deficiency, relating rather to priorities, and the presumed (compile-time) cost of traversing the ad-hoc heap allocations of node-based containers such as *std::list* and *std::map*. This is in the context of copying (cloning) per-thread data ranges during parallel mapping or convolution operations; as discussed in Section .

An example usage of a *constexpr* parallel *std::for_each* from ClangOz's custom Algorithms library can be found in Listing 1. *std::execution::par* is a simple, 1-byte, pre-constructed *std::execution::parallel_policy* object, triggering the compile-time parallel evaluation of the loop contained within the function's definition. The 4th argument, a C++ lambda function, is then executed in parallel on the elements of the *std::array* object.

```
constexpr auto f() {
  std::array<int, 4> arr{1, 2, 3, 4};
  std::for_each(std::execution::par,
                arr.begin(), arr.end(),
                [](int &i) { i++; });
  return arr;
}
```

Listing 1: ClangOz's modified libstdc++ supports parallel compile-time algorithm evaluation via the standard *std::execution::par* policy object; allowing users to avoid low-level compiler intrinsics.

The parallelisation process takes place within Clang's constant expression evaluator. The constant expression evaluation executes within the frontend, usually during the semantic analysis stage; either when generating the abstract syntax tree (AST), or during later code generation.

The constant expression evaluator attempts constant folding on expressions stored in AST nodes; collapsing them into a value or values during compilation by processing the expression. The values are calculated and stored using Clang's *APValue* class. This class holds constant data of arbitrary bit-widths for several C++ value types; including *float*, *integer* and arrays.

Figure 1 may help illustrate the placement of constant evaluation within Clang's frontend. *EvaluateExpr* is invoked when a constant expression requires evaluation, and may subsequently call the *EvaluateStmt* function of Clang's ExprConstant.cpp. *EvaluateStmt* is responsible for evaluating statement-level constructs during the constant expression evaluation process. It recursively processes various statement types, such as conditional statements, loops, and return statements, to determine their constant values if they can be wholly resolved at compile time. Its loop handling is where most of ClangOz's modifications are located.

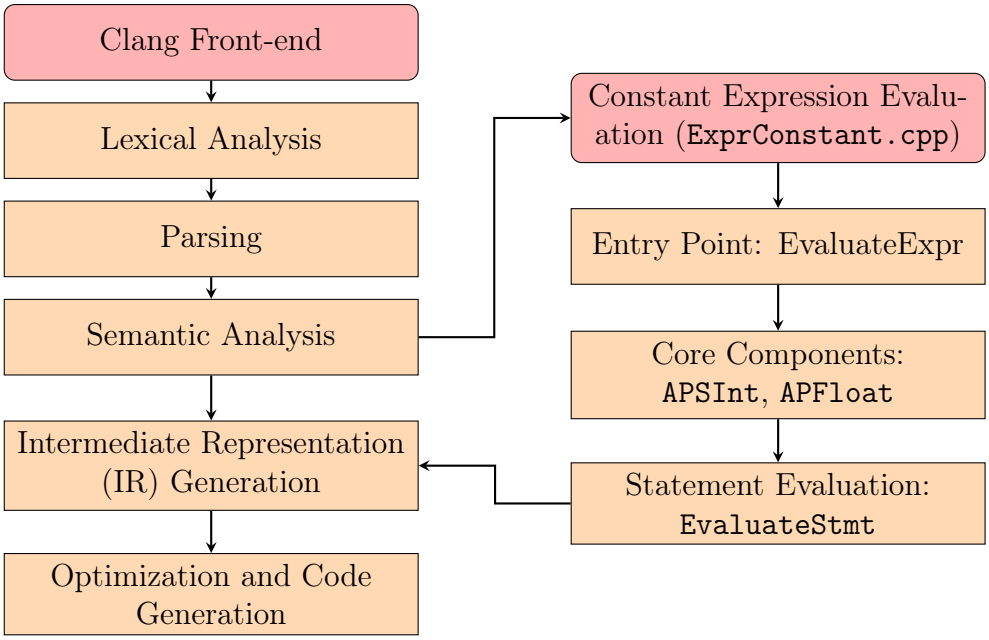

Figure 1: Constant evaluation within the Clang Frontend

Manipulation of *APValue* objects is pivotal to the parallelisation process, and in particular those that are *LValue*s. Generally, *LValue*s are locators for objects. An *LValue* can contain either the path from a complete object to

its subobject; or a memory address offset. These are important as the majority of the parallelised standard C++ library algorithms use *iterators*: an idiomatic abstraction over the traversal strategy of each container. The parallelisation process manipulates and gathers information from these iterators; with pointers a common form.

Clang's *CallStackFrame* and *EvalInfo* classes are also integral. The former acts as a call stack for the constant expression evaluator; maintaining information for the current call stack frame, and tracking the arguments passed to the frame and the temporaries that reside within it. Alongside, a pointer to the preceding frame in the stack is also stored, to facilitate backwards traversal. The *EvalInfo* class maintains information about the expression being evaluated, including the *CallStackFrame*. These classes maintain most of the evaluator's state during evaluation.

Two Clang AST components that are useful for the parallelisation process are the *Expr* and *Decl* family of classes. The former maintains information about types of expressions; for example *CallExpr* maintains information about function invocations. The latter tracks declarations or definitions of different language constructs; for example information on each function definition is stored within a *FunctionDecl*.

3.1. The Parallelisation Process

The parallelisation process consists of four phases (see Figure 2). The first of these is *verification*, confirming that each *for* loop is encountered at function scope within a *constexpr* context.

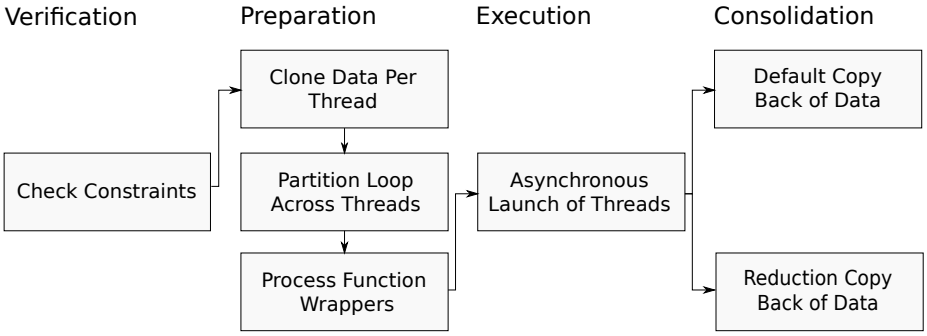

Figure 2: Compilation Phases involved in the Parallelisation Process

The second phase is *preparation*, where the intrinsics are processed; local data is prepared for each thread; and the loop space is partitioned. This

phase involves creating a clone of the *EvalInfo* object per thread, as well as each of the *CallStackFrame*s it contains. The *APValue*s that reside in each *CallStackFrame* are also cloned; representing both dynamically and statically allocated data.

After the data has been cloned the partitioning process begins, using static loop partitioning to divide the work across multiple threads. If the data cannot be divided evenly across threads which are maintained by a single thread pool; any excess work is given to the final thread. This partitioning process is part of the *LoopIntrinsicGatherer* class, an addition to ClangOz that implements the functionality for handling the intrinsics, cloning data and reducing data. The partitioning is reliant on the *__BeginEndIteratorPair* or *__PartitionUsingIndex* intrinsic being used by the creator of the function to specify the loop bounds. These and other intrinsics are discussed further in Section 3.2.

The intrinsics are discovered prior to the partitioning process by traversing the function body containing the loop, statement by statement, checking the name of each function called against the list of intrinsic names. This is done by making the *LoopIntrinsicGatherer* a child of Clang's *ConstStmtVisitor* which recursively visits a *Stmt*, breaking it into its constituent *Stmt* types. Each *Stmt* in the body of the *FunctionDecl* is then iterated over, and passed to the recursive visitor, which then searches for *CallExpr* nodes to verify and process.

Each thread has copies of the variables defining the loop bounds; provided in the initialisation, condition and iteration statements. Dividing the workload across threads requires offsetting the underlying *APValue*s of these variables; and in particular those found in the loop's condition statement, which help to define its range. In the case of the standard library algorithms, the loop conditions are equality checks; for example, comparing the start and end pointers from a container to check that they are not equal. It is possible to offset the pointer's *APValue* to point to the start and end of the loop partition for each thread, effectively segmenting the loop.

To calculate the appropriate size of each partition, and the amount required to offset the loop's start and end by, the size of the iteration space must be calculated. This distance can then be divided by the number of partitions provisioned. For loop bounds defined by integer values, this is straightforward. With pointers, the size of the container's element type is required; and memory addresses of a contiguous container are traversed using an offset based on the size of the element type. Calculating the distance

requires utilising this size, to convert from a memory address to an integral number representing the loop's range. This can then be used to calculate the offset for each partition, before each pointer is offset by the appropriate amount. Only containers of contiguous data are supported, as partitioning non-contiguous data involving arbitrary memory locations is non-trivial, and time intensive.

After the work has been distributed, the third phase begins: the *execution* phase. Tasks, encapsulated as C++ function objects (often lambda functions), are launched asynchronously, and then a *wait* for completion is issued. The parallelised task contains the *constexpr* evaluation of the body of the loop. The initialisation and destruction steps in the loop's evaluation are executed sequentially, and occur once on loop entry and exit. The task itself does not deviate from the original sequential algorithm.

The final phase after thread completion is *consolidation*. This phase focuses on synchronising thread data back into the main process's *CallStackFrame* and *EvalInfo*, allowing sequential evaluation to continue. This is done in two steps, the first copies the cloned data back into its original location. The second step involves an optional reduction, and is controlled by the *__ReduceVariable* intrinsic discussed in Section 3.2. Data that is marked for reduction by *__ReduceVariable* skips the first step.

Data which has been cloned, is split into two components before being copied back. The second component is specific to array data, the first is for everything else. A primary thread is selected to copy data for the non-array component, which is dependent on an *EvalStmtResult* object returned by each parallel task. This *EvalStmtResult* is a Clang enumerator that contains different evaluation result flags for each statement type. Each returned *EvalStmtResult* is checked: if all return successfully, then the final thread is selected as the primary thread. As the whole loop range was iterated across, the newest values should be contained within the final partition space. In other cases, where threads complete early, perhaps due to encountering *return* or *break* statements, the first thread that signalled early completion is selected. This ensures that values in later partitions are ignored, as they would not be processed when executing the loop sequentially.

The re-synchronisation of arrays is done by determining which elements have been written to by each thread and then copying these elements from the respective clone, to the original. This does not factor in alteration of the same array element by multiple threads.

Applying reductions can be thought of as a special case of the first step

which can be requested by a user through the *__ReduceVariable* intrinsic when a more complex data synchronisation method is required. There are several different types of reduction possible, and these are discussed in Section 3.2.

3.2. The Intrinsic Functions

The compiler intrinsics are invoked much as standard C++ functions, and communicate to the compiler how a loop is to be parallelised. They are implemented as functions rather than Clang intrinsics as it simplified modifications to the parallelisation process. This use of an API of intrinsics has much in common with OpenMP [42] and other directive-based programming paradigms.

The intrinsics required to describe the parallelisation of a loop should be placed prior to the loop within a function that meets the aforementioned constraints. There are four different intrinsic functions used for parallelisation, with descriptions listed in Table A.1. The intrinsics have no body and are "no-ops" at runtime with a trivial overhead during compilation. They are defined as function templates so that they can be used with a variety of different types. The name of the intrinsics are prefixed with double underscores to avoid conflicts with user-defined functions. The parameters of each intrinsic allows users to pass important information to the compiler.

__BeginEndIteratorPair and *__PartitionUsingIndex* indicate to partition iterations of the loop across multiple threads based on the loop bounds indicated by their arguments. The former was designed with the use of C++ standard library containers and algorithms in mind, which make use of *begin* and *end* iterators to mark the range of loops. The latter was designed with numeric loop conditions in mind and takes an extra parameter indicating the relational operator used within the loop's condition clause.

__IteratorLoopStep indicates that a pointer based index is bound to the loop's step. Clones of the index are offset by the number of loop steps taken by the loop in the thread partition at its start point. The offset is calculated by mutating the index by the number of loop steps taken by the C++ operator indicated by the *OpTy* argument. This keeps the bound value synchronised with the loop across all threads, and is used for indices not used within *__BeginEndIteratorPair*.

The intrinsic *__ReduceVariable* helps to denote how a container or value should be reduced when the launched threads are joined. Three reduction

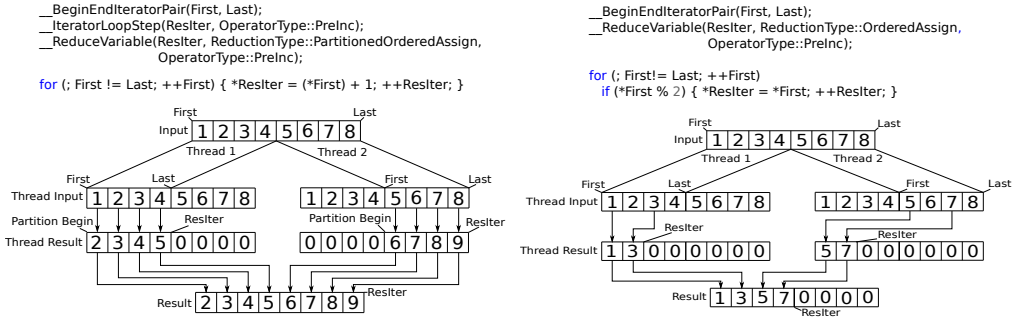

Figure 3: Example PartitionedOrderedAssign (Left) and OrderedAssign (Right) Reduction With Two Threads

types are supported: *PartitionedOrderedAssign*, *OrderedAssign* and *Accumulate*. *Accumulate* is used in conjunction with an accumulator variable, ensuring that the local result from each thread is combined using the specified operator to obtain a final value. *PartitionedOrderedAssign* is intended for use with containers, and its operation is illustrated on the left in Figure 3. This reduction assigns elements to the original container in order, where each element is taken from the starting offset in each thread partition to its final offset on thread completion. This allows appropriate collapse of data as threads are working on local copies of data rather than shared data. *OrderedAssign* (on the right of Figure 3) is similar to *PartitionedOrderedAssign*, although it is used with containers that have not been modified in lock step with the loop. *OrderedAssign* assigns elements to the original container in order, where each element is taken from the initial offset of each cloned container to its final offset within its partition.

3.3. An Example Constexpr Parallel Function

Within the modified standard C++ Algorithms and Numerics library included in the libstdc++ fork created for ClangOz, 30 functions have been *constexpr* parallelised. In Listing 2 *std::for_each* is shown as an example of how a function can be *constexpr* parallelised. The function takes an execution policy as its first parameter which will be verified by the compiler before it attempts parallelisation. In this example there is also an alias for an *std::enable_if* check, which ensures the correct execution policy is used in conjunction with this variation of *std::for_each*. Alternatively, the compiler will select a more apt function if one exists, or issue an error message.

```
template <class _ExecutionPolicy, class _FwdIterator, class _Function>
constexpr
__enable_if_constexpr_par_execution_policy_t<_ExecutionPolicy, void>
for_each(_ExecutionPolicy&& __exec, _FwdIterator __first,
         _FwdIterator __last, _Function __f)
{
    __BeginEndIteratorPair(__first, __last);
    __ReduceVariable(__first, PartitionedOrderedAssign, PreInc);

    for (; __first != __last; ++__first)
        __f(*__first);
}
```

Listing 2: The *constexpr* parallel *std::for_each* implementation

Once the policy has been verified, each of the intrinsics is processed; and in this example there are only two. The first, *__BeginEndIteratorPair* defines the loop's range which the parallelisation process uses to partition the loop across multiple threads. In this case the range is delimited by the arguments *__first* and *__last*. The second intrinsic *__ReduceVariable* states that a *PartitionedOrderedAssign* should be performed on the data pointed to by the argument *__first*. *OperatorType::PreInc* indicates which operator to use when traversing the data, allowing the compiler to correctly reduce the data.

4. Runtime and Compile-time Libraries

The first library dependency of most programs is included with the invoked compiler: the standard library. The C++ language and library are specified together, every few years, within each update of the C++ standard document. The latest version was ratified and published by ISO in December 2020 [17]. C++20 saw the *constexpr* specifier added to a majority of the function templates from the Algorithms and Numerics libraries; and support for these across the three main standard library distributions (libstdc++, libc++ and Microsoft STL) came quickly. Intrinsically, such function *templates* are statically polymorphic, and one can quickly test their compile-time functionality using containers which allocate their memory resources statically (e.g. *T[N]* or *std::array*).

C++20 also incorporates seven proposals, a series of "relaxations of constexpr restrictions", including provision for *new-expressions* within constant expressions; so enabling the use of dynamic memory allocation within compile time calculations.

While end users can therefore then utilise pointers, and compile-time dynamic memory allocation, in their own code *directly*, the feature naturally also allows the development of safer, high-level dynamically-sized containers. Such motivation is apparent in C++20's adoption of two library proposals which add support for the *constexpr* specifier to all methods of the *std::vector* and *std::string* class templates.

All three main standard library distributions (libstdc++, libc++ and Microsoft STL), today support *constexpr* versions of the *std::vector* or *std::string* class templates. Yet, as we prepare to work with such convenient abstractions, it is worth highlighting the implications of the following: compile-time allocations must be deallocated before the end of the containing constant expression. Consequently, as readily as the destructor of an *std::vector* or *std::string* object will deallocate the memory it owns, so will a *constexpr*-qualified declaration of a variable of either type, within code *capable of runtime execution*, produce a compilation error. The rationale behind the new term *transient* for C++20's *constexpr* memory allocation is thus apparent. It follows that the *result* object of the elaborate constant expressions or programs under discussion here, must not itself rely on dynamic memory allocation.

Yet real-world codes, whether executing at compile-time or runtime, should expect to utilise far more than just *std::string* and *std::vector*. The C++23 specification relaxes some (more) *constexpr* restrictions in the language [43, 44, 45]; while the corresponding library specification adds near full *constexpr* support to *std::unique_ptr* [46], *std::bitset* [47], *std::optional* and *std::variant* [48]; with ad-hoc *constexpr* coverage for individual functions [49, 50], and the non-transcendental components of the numeric library's *cmath* component [51]. There are though still scores of classes and functions with no *constexpr* provision (i.e. the majority). All of our benchmark programs utilise far more *constexpr* functions than those specified by either the C++23 standard, or the C++26 working draft.

4.1. C'est 2: An Extended C++ Standard Library

We now introduce the *C'est 2* library. C'est 2 presents an API intentionally similar to the C++26 draft standard library, supporting additional

constexpr annotations on the familiar class and function definitions therein. A predecessor of this entirely new library (C'est [52]) was developed, used, and briefly discussed in our prior research [6, 7]. C'est was structured in a simple, portable, though limiting form: as an auxiliary component to an existing C++ standard library implementation. A significant restriction on this approach meant that, to avoid compiler naming clashes, all additional *constexpr*-enabled functions had to be defined within a namespace distinct from the standard C++ library's *std*. After the library name, the namespace was spelled *cest*; and hence one would use *cest::deque* rather than *std::deque* for example. Yet, while C++ namespace alias commands can help assuage a user's distaste for a new library dependency, the engineering toll inevitably rises along with the size of each software project considering a *constexpr* port.

In this work, we present a fresh alternative for those interested in extended *constexpr* adoption within a C++ standard library. The C'est 2 *constexpr* "runtime" library [53] is implemented as a fork of GCC's libstdc++ [8]; the standard library shipped with most Linux distributions; and used by the Clang compiler as a default on those platforms.

In the usage scenarios examined in Section 5 we link (when necessary) against a build of an *unmodified* release of the libstdc++ source code; stored within the *master* branch of our forked libstdc++ source code repository [53]. The vital *constexpr* definitions exist within a header-only library, as the source code for each function must be available during compilation to permit constant evaluation. These *constexpr* function definitions, within modified C++ standard headers are then selected by the compiler during constant evaluation, instead of the definitions stored within the libstdc++ shared object binary file.

The library supports useful *constexpr* functionality not otherwise found within the C++ standard library specification, or any other standard library implementation. We include *constexpr* versions of the standard library's *forward_list*, *list*, *set*, *map*, *queue*, *deque*, *shared_ptr* and *function*. We also permit I/O commands to compile within constant expressions, albeit without the traditional side effects (e.g. *std::cout « "Hello World"*). This is primarily to support execution of existing code bases within both compile-time or run-time contexts. The code listing in Appendix B provides a concise example of the range of *constexpr* functions and classes available.

5. Parallelism Benchmarking

Seven *constexpr* programs based mainly on existing benchmarks or programs were created to test the performance of the *constexpr* parallelism implementation. The benchmarks were chosen as they have already proven to be parallelisable in a way that lends itself to the *constexpr* paralleliser. For example, complex orchestration of tasks or non-blocking tasks are not within the scope of the project; data parallelism is our focus. The programs are also selected as being capable of conversion to a *constexpr* program, in preference to those with language features which are incompatible with constant expressions; for example embedded assembly language instructions.

All benchmarks are parallelised using static partitioning. The partition sizes are selected by the compiler based on the number of threads made available and the size of each loop's range. The parallelised regions are indicated by an invocation of the higher-order *std::for_each* or *std::transform_reduce* function templates, which have been adapted to support *constexpr* parallelisation. The *std::for_each* invokes a unary function, provided by the caller as the final argument, on each element; while *std::transform_reduce* applies a given unary function to each element, and reduces the results along with a given initial value over a given binary function. Overloads of the *std::for_each* and *std::transform_reduce* function templates are the only parallelised algorithms used within these benchmarks. It is worth noting that the main cost of the parallelisation algorithm is the cloning of data between thread partitions. Not only do the C++ iterators require cloning but so does any data used within the algorithm, such as any lambda captures, temporaries in use, or data being worked upon by the function; for example an array pointed to by the iterators.

The detailed performance data gathered for each benchmark is displayed using two types of line graph. The first compares serial and parallelised execution times within instrumented code regions. Each of the plots in these graphs are calculated by averaging six separate runs of each of the variations of the benchmarks. The second line graph type examines performance scaling, comparing the *speedup* when using different numbers of threads against the ideal speedup on different problem sizes for the benchmark. The ideal speedup rises in direct proportion to the number of threads used. The remaining benchmark data is the difference in performance between the serial and parallel data sets. An aggregate graph that presents the speedups for all benchmarks is also included to allow a holistic analysis.

5.1. Benchmark Programs Overview

Two benchmarks are taken from the Princeton Application Repository for Shared-Memory Computers (PARSEC) [54]. The PARSEC suite contains several multi-threaded programs that explore different workloads on shared memory architectures. The Swaptions and BlackScholes examples were selected. Both models process financial data, however they use different methods of calculation. Swaptions makes use of Monte Carlo simulation and BlackScholes uses a partial differential equation.

The N-Body problem and Mandelbrot benchmarks are based on existing solutions provided to The Computer Language Benchmarks Game [55]. These are micro-benchmarks with the goal of testing performance of different programming languages as opposed to directly testing parallel performance. These benchmarks are interesting in this case as they both require *multiple* fork-join threading regions to attain results as opposed to the single fork-join region of Swaptions and BlackScholes.

The Sobel edge detection edge benchmark is based on an example of the heterogeneous parallel programming model: Khronos SYCL [56]. SYCL is a C++ programming model which aims to simplify programming on heterogeneous architectures. The benchmark provided an opportunity to test the feasibility of using the *constexpr* paralleliser within a larger application. The *constexpr* paralleliser replaces the underlying programming model being abstracted by SYCL, facilitating a compile-time parallel implementation. The SYCL benchmark performs the Sobel edge detection algorithm on an input image. It is similar to the N-Body and Mandelbrot examples as it requires multiple fork-join threading regions; with no iteration, though the tasks are more computationally expensive.

To ensure that the performance of parallel reduction operations are also considered, two new benchmarks are introduced. A suitable function template supporting a parallel execution policy parameter comes from the C++ standard Numerics library: *std::transform_reduce*. This function template is convenient in that both of our benchmarks involve a *map* operation, followed immediately by a *reduce* operation. As *std::transform_reduce* can rely on one fork-join threading region; the overhead of two is avoided. The first benchmark counts the number of space-delimited words contained within a file. This benchmark relies on an overload of *std::transform_reduce* which accommodates two iterator pairs, both of which represent a view upon the same data; an approach common in the array expressions of High Performance

Fortran. The newly created Word Count benchmark program is directly inspired by part of Bryce Lelbach's CppCon16 presentation [57]. The second reduction benchmark is a 3D heat equation solver, derived from a suite of runtime benchmarks supporting a recent performance portability paper [58]; with each time step involving a *std::for_each* call for halo exchange and finite difference methods; concluding with a single call to *std::transform_reduce* to compare the numerical and analytical solutions.

5.2. Timing Constant Evaluation

Performance of the *constexpr* paralleliser is analysed by measuring the duration of a parallelised constant expression's evaluation, and comparing against that of the original serial evaluation on each of the seven benchmarks. The benchmarks are tested with different numbers of threads using an Intel Core i9-13900K CPU, containing 8 *performance cores*, with support for 16 hardware threads via simultaneous multi-threading (SMT; or Intel's Hyper-Threading); and 16 *efficient cores*. The base/max turbo frequencies of the effiency and performance cores respectively are 2.20GHz/4.30GHz and 3.00GHz/5.40GHz. The benchmarks were run under Ubuntu 23.10; and executed using two, four, eight and sixteen threads.

Time is measured from the beginning of a parallel region to the end of a parallel region. Rather than timing the length of the entire program, we measure only the regions of interest. The same location is measured for the serial execution. As an unmodified C++ compiler does not have any mechanism to measure or record times during constant evaluation, three intrinsics were developed within ClangOz to facilitate such benchmarking: *__GetTimeStampStart*, *__GetTimeStampEnd*, and *__PrintTimeStamp*.

__GetTimeStampStart stores an initial time point/stamp within the compiler when the *constexpr* evaluator passes through it; its partner, *__GetTimeStampEnd*, stores a second time stamp, delimiting the region of interest in time. The third intrinsic, *__PrintTimeStamp*, calculates the difference between the two time stamps, and prints it to the console's standard output. This simple interface of compiler timing intrinsics is flexible, and entirely suited to the domain. Unlike benchmark systems which relate to template metaprogramming [59], we are not here concerned specifically with C++ template instantiation; instead we proceed, as with most *constexpr* software development, in much the same way as we would when developing a program for runtime execution. In the example shown in Listing 3, the

```
constexpr void timing_example()
{
  std::array<int, N> id_range{};

  __GetTimeStampStart();
  std::iota(std::execution::par,
            id_range.begin(), id_range.end(), 0);
  __GetTimeStampEnd();

  __PrintTimeStamp();
}
```

Listing 3: Timing a parallel *std::iota* call using ClangOz compiler intrinsics

user has decided to measure the parallel constant evaluation of the *std::iota* function template call; but not the time to create the *std::array* object.

It is worth noting that the same compiler is used for both the parallel and serial tests, as the intrinsics are needed for timing alongside a modified standard library implementation. This has a minor impact on the measurements for both implementations as the timing functionality requires checking the intrinsics' names, every time a function is considered for parallel execution. To determine if the parallel code path should be executed within the compiler, the verification phase discussed in Section 3.1 must be processed, which also adds an extra performance cost when evaluating the original serial implementation.

5.3. Mandelbrot

The Mandelbrot benchmark consists of three main areas of computation that are executed in parallel using *std::for_each*. A 2D array of complex values (a class containing two 64-bit floats), corresponding with each output pixel, is initialised based on its coordinate. Subsequently, the main Mandelbrot computation uses the naïve escape time algorithm. This algorithm loops over each complex value and performs a repeating calculation until an escape condition is met (limited to a maximum of 128 iterations). The final value generated after the escape condition has been met is the colour of the pixel which is assigned to an array of integers representing the final image.

The graphs in Figure 4 show that the increase in data size gradually pro-

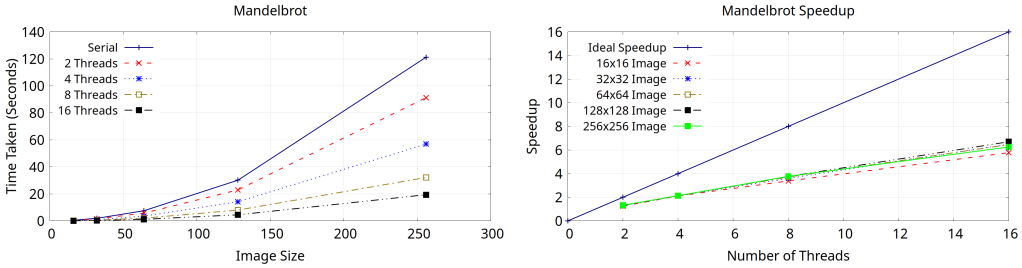

Figure 4: Compilation Times and Speedups for the Mandelbrot Benchmark

gresses towards higher polynomial growth as the number of threads diminish. With more threads, the increase in data size has less impact on compilation time. In our earlier work [7] the computation involved two separate parallel regions, requiring two thread group launches. The two parallel loops have since been fused; though this has had minimal impact on the performance. The speedup graph reveals tolerable scaling. Though with a thread count of 8 all problem sizes have a speedup closer to half of that, it is encouraging that performance does improve a little as SMT is engaged for 16 threads.

5.4. BlackScholes

BlackScholes has two areas of computation, requiring two separate loops. The first is the main kernel which computes the Black-Scholes equation, and uses a parallelised *std::for_each* call; the second verifies the results from the first computation in serial. The main data that requires cloning in this benchmark is a 1D array containing a C-style *struct* for each input that owns 9 floating-point values; parsed from an external header data file using a *#include* preprocessor macro.

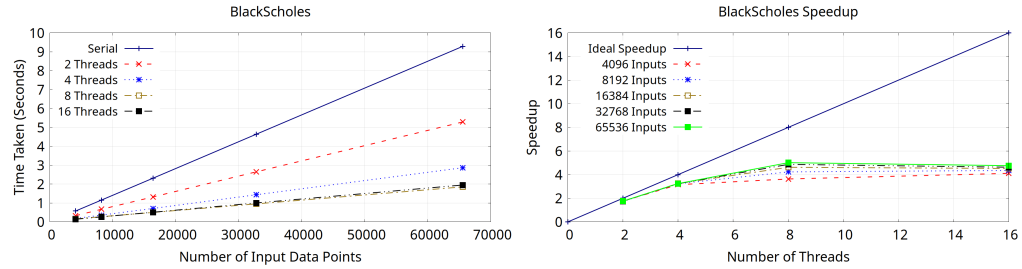

Figure 5: Compilation Times and Speedups for the BlackScholes Benchmark

The BlackScholes data in Figure 5 shows promising performance increases when utilising both two and four threads, but begins to wane with larger thread counts. The speedup graph on the right shows an ebb in the speedup value for all data sizes, as the thread count moves from eight to sixteen. A possible explanation here is that simultaneous multi-threading is being used on the eight performance cores, and the execution units on each core are already maximised.

5.5. N-Body

The N-Body benchmark has two parallel regions: the advance of the particle system; and the position and velocity update. This means with more iterations of the system, more fork-join thread launches occur. To allow a range of body counts, the number of iterations is restricted to 32; while still avoiding Clang's limits on constant evaluation step quantities. The main data cloned within the benchmark are the body attributes: structures containing seven 64-bit floats stored contiguously within an array. The number of bodies is the parameter that is varied within this benchmark.

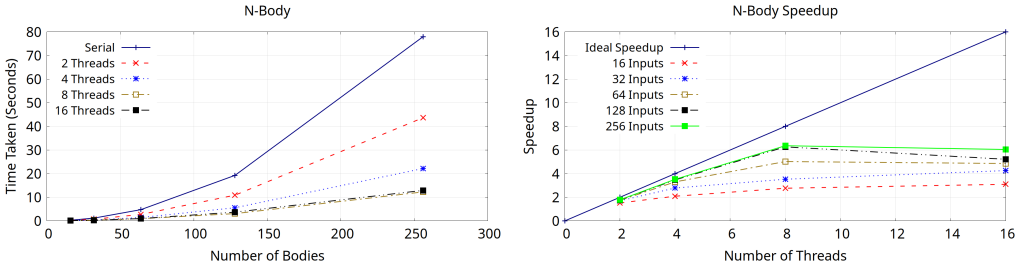

Figure 6: Compilation Times and Speedups for the N-Body Benchmark

The graphs in Figure 6 show that increasing the number of threads again outperforms the serial implementation, thought eight threads is the closest to the ideal speedup achieved within this benchmark, with larger body numbers also aligned with better performance. This is likely due to the cost of cloning having an adverse impact on smaller workloads. As with the BlackScholes benchmark, there is no improvement in speedup as the number of cores is increased from eight to sixteen (again likely due to SMT's limitations).

5.6. Swaptions

One parallel region within the Swaptions benchmark encompasses the entire algorithm, calculating the pricing of a portfolio of swap options. The

swaption data being cloned within the benchmark is comprised of two 1-D arrays containing contiguous data. The 1-D arrays are comprised of C-style *struct*s, which contain floating point data. Whilst the Monte Carlo simulation in use is stochastic, this implementation uses a fixed seed for the random number generation.

The performance of this benchmark is linear, both in the number of swaptions, and the number of trials; with the number of swaptions varied across the different benchmark runs. In an earlier evaluation of this benchmark, we were compelled to use a value of 2000 for the number of trials; a configuration which unfortunately forced us to vary across a relatively small number of swaptions for the performance analysis; with extremely long compilation times ceding to Clang's hard limits on constant evaluation step counts. Attempts to use smaller iteration counts for the accumulation stage, corresponding to a lower number of trials, led to a handful of problematic arithmetic operations involving NaN ("Not a Number") values. While these are ignored by default during *runtime* execution of this benchmark, arithmetic involving NaNs leads to a compilation error during constant evaluation[3]. We have since been able to modify the benchmark. The creation of each NaN is not of itself a problem, and so we now simply identify when one is created, and avoid performing any arithmetic on it. This has allowed us to investigate a far greater number of swaptions in the current performance analysis.

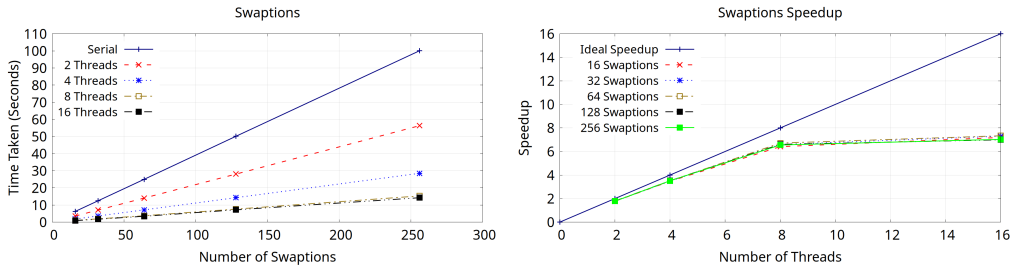

Figure 7: Compilation Times and Speedups for the Swaptions Benchmark

The swaptions are evenly partitioned across the threads and are the locus of computation. It can be seen in Figure 7 that as the number of swaptions

[3]Annex F of the C standard specifies that IEC559 floating point exceptions encountered at compile time should not stop compilation; though adoption of this point in C++ compilers has been slow.

can now be perfectly divided across the threads, performance is good; with the speedup graph scaling close to the ideal, at least until the stable point of eight threads. Adding eight more threads, again likely via the SMT eight performance cores, adds little benefit.

5.7. SYCL Edge Detection

The SYCL benchmark is broken up into five areas of parallelism, each hidden behind the SYCL API's *parallel_for* construct; itself implemented upon our *constexpr* parallel *std::for_each*. Nesting the *std::for_each* comes at a cost, as it lies several layers down the function call stack every time threads are launched. This can present a performance challenge to the parallelisation process, as it must expend resources cloning each call stack frame above the *std::for_each* for each thread. This is required even if data from the prior frames is unused, albeit in this case the cloned call stack frame is empty. A further cost is the cloning of C++ lambda functions, which define the tasks to be executed within the SYCL programming model. Each area of parallelism computes a component of the final output, performing the convolution and filter application on the input image. The data that requires cloning in this benchmark are two 1-D arrays containing unsigned integers which are contained within another SYCL construct.

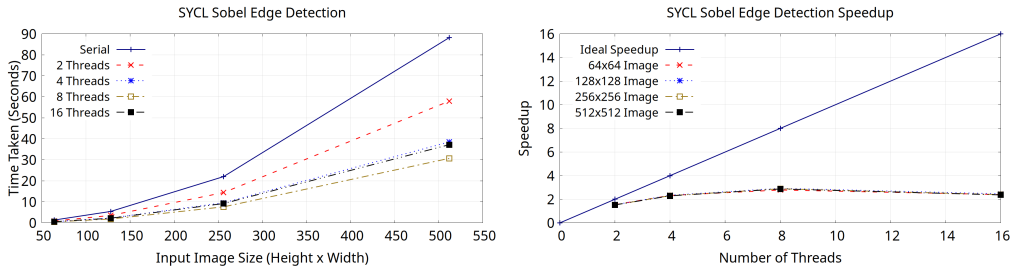

Figure 8: Compilation Times and Speedups for the SYCL Edge Detection Benchmark

The data in Figure 8 shows that over-subscription of threads has diminishing returns. Furthermore, while we might hope for larger speed-ups on larger data sizes, all image sizes in fact induce a similar speed-up. The extra call stack depth and cost of cloning the lambdas and their captures are likely the cause of these effects at larger thread counts; and generating threads that are not being instantly utilised comes at a greater cost. A distinguishing feature of this benchmark in performance terms is the involvement of a larger

number of parallel regions (five); each of which will have overheads relating to thread launches, and data cloning.

5.8. Word Count

The Word Count benchmark program largely consists of a common *#include* directive to load the contents of a plain text file provided as input; followed by a single call to *std::transform_reduce*. A C++ runtime program was created to generate files containing words in random order, of a specified quantity. A simple existing bash script then modified the contents of each file to adopt the syntax of a C++ raw string literal; subsequently used to initialise a *std::string* within a *constexpr* function. The *std::transform_reduce* overload then uses two *begin* iterators, starting at each string's first and second character respectively. The *map* operation here is akin to a *zip*, producing an intermediate container of boolean values, set *true* at the start of each word. A *std::plus* function object is then used for the reduction step: summing the boolean values as integers, and therefore counting the space-delimited words within a file. As usual, initialisation, here via the *#include*, is not timed.

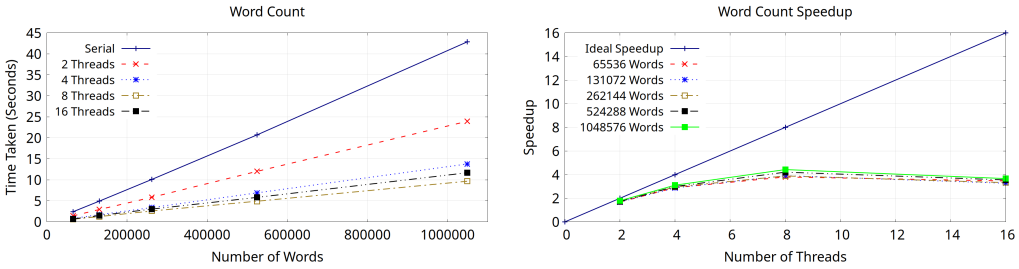

Figure 9: Compilation Times and Speedups for the Word Count Benchmark

As the Word Count benchmark consists almost entirely of one call to *std::transform_reduce*, the algorithm has a complexity of $O(n)$. Indeed, the expected linear structure of the compilation time measurement results are clearly apparent on the left graph of Figure 9. Performance scaling, shown on the right, adopts a familiar pattern: attaining a peak of relative performance over the scalar version, at the point where the largest number of physical cores are utilised. After that, with 16 threads running on 8 physical cores via SMT, performance falls relative to the serial version.

5.9. 3D Heat Solver

Most of the 3D Heat Solver execution is occupied by linear iterations over the data; with the data size varying as the cube of the problem size provided as input. Each time step involves a parallelised call to *std::for_each* for halo exchange and the forward time-centered space (FTCS) finite difference method; concluding with a single verifying call to *std::transform_reduce* to compare the numerical and analytical solutions. With each iteration causing threads to complete a potentially expensive fork-join, and this clearly a computationally intensive benchmark, the iteration count was set at 100 (the original paper [58] uses 1000). This allows a greater range of problem sizes to be examined. Note also that as this benchmark relies on a number of transcendental functions, the *constexpr* GCE-Math library [10] is utilised.

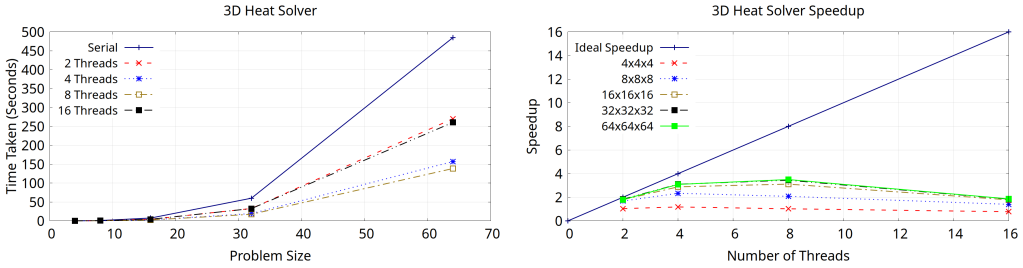

Figure 10: Compilation Times and Speedups for the 3D Heat Solver Benchmark

Performance results for the 3D Heat Solver are presented in Figure 10. Compilation times on the left track a cubic logarithmic response which presents a straightforward pattern for each thread count. On the right however, scaling remains low, with a maximum speedup of almost 3.4 at the point where 8 physical cores are utilised, before dropping further. While the low scaling of the bottom-most (4^3) data line could be attributed to its small size, the line for the largest size (64^3) has a similar trajectory. Ultimately the iterative structure of this benchmark leads to 100 thread fork-join instances, as produced by each iteration's *std::for_each* invocation.

5.10. Benchmark Comparison

The speedup graph in Figure 11 compares the largest variations of each benchmark against each other and indicates which benchmarks achieved the best performance scaling through parallelisation. The Swaptions benchmark

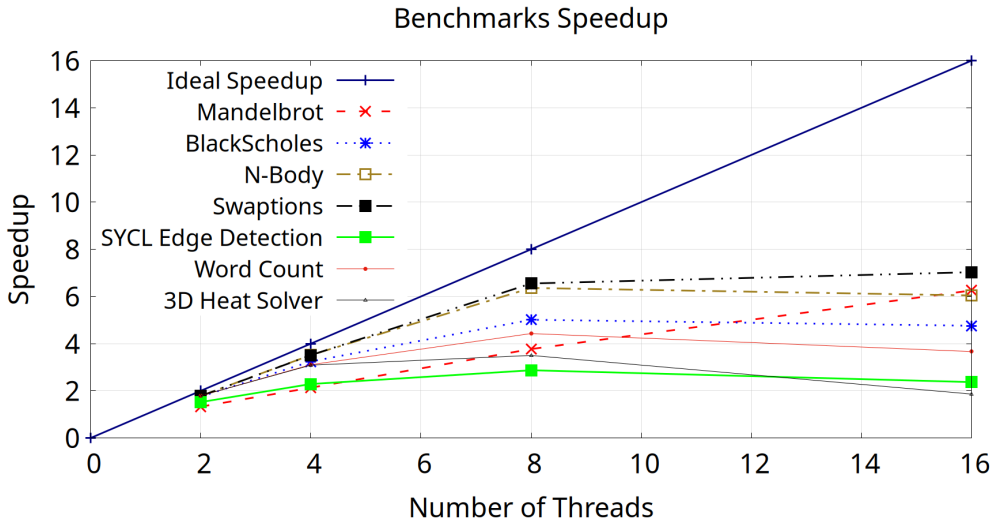

Figure 11: Speedup Graph Comparison of all Benchmark Compilation Times

attains the highest speedup at the highest thread count of sixteen; and respectable scaling at eight threads and lower. For the N-Body benchmark, the trivial size and simplicity of the data that requires cloning works in its favour, and demonstrates comparable scaling to the Swaptions benchmark up to and including eight threads. After that however, with sixteen threads, the utilisation of SMT by the LLVM thread pool seems to work against it. Until eight threads, the Mandelbrot benchmark is farther from the ideal speedup than N-Body; yet its regular linear progression sees it continue to climb, and attain a similar speedup at 16 threads. The possibility of multiple thread launches causing cloning to have a negative impact is highlighted by BlackScholes performing better than the SYCL Edge Detection program, despite having a similar amount of data to clone per parallel region, but less parallel regions overall. With sixteen threads, SMT should allow two threads to run efficiently on each of the eight performance cores, but only the Mandelbrot benchmark show a noteworthy reduction in execution time at the largest thread count. The SYCL Edge Detection and 3D Heat Solver benchmarks are distinguished by their poor scaling. While more *physical* cores always corresponds to a slight improvement in absolute performance, when the amount of cores is accounted for, as in the speedup data, the scaling curve for SYCL Edge Detection appears almost flat; while the smallest

data set for the 3D Heat Solver actually displays a slow-down.

In summary, two performance curves stand out. The Mandelbrot benchmark results are notable in that we do not see its speedup drop as the others do as it transitions from 8 to 16 cores. Also, the 3D Heat Solver is distinct in its relatively poor scaling throughout.

The results nevertheless indicate that the parallelisation of *for* loops during constant evaluation can lead to notable speedups when a large portion of the program conforms reasonably to a small set of loops. However, the cost of cloning and partitioning of data comes with significant costs. It is plausible that performance could be increased by removing the need for cloning in cases where it should not be required. For example, in contrast to convolutions, mapping operations typically do not require temporary allocations of data while processing ad-hoc sections of C++ container objects. Such an approach could simply overwrite existing data, and seems especially plausible given that the low-level intrinsics API we use, does convey such information. So too, data that is not required for the execution of the *constexpr* function containing the *parallel* loop could also be elided from the current cloning process, which could have a large impact on benchmarks that contain a significant amount of data unrelated to the parallel invocation. A simple form of workload balancing may also yield reasonable results in certain circumstances where the majority of the work is not perfectly divisible by the number of threads in use. Whilst this is not seen in the performance analysis within the paper, there is likely an opportunity for improvement over the current implementation that could allow a performance increase.

5.11. Cross-vendor Compilation Timing

The performance analyses explained above, measure the time of each benchmark program's kernel; avoiding the performance impact of initialisation, and focusing instead on those regions which are capable of parallelisation. This is a conventional approach in high-performance computing and after all, only the ClangOz compiler is capable of emitting precise timing information *during* constant evaluation; as described in Section 5.2. Yet questions remain as to the overheads of the new compile-time parallel evaluation feature; and of whether a completely different C++ compiler may already offer performance outperforming its parallel evaluation. Consequently, the first of two new experiments is described and analysed below; involving

unmodified GCC and Clang compilers[4].

In this experiment, the overall build time (compilation and linking) is measured, for each of the seven benchmark programs described earlier. Three different compilers are utilised, with one applied in two different configurations. The latest version of GCC (version 14.0.1) included with the current Ubuntu 24.04 distribution package is utilised as a reference; with the times of the other three compiler configurations expressed as a relative percentage of its times. A similar Ubuntu package version of Clang (version 18.1.3) is used. Lastly, our modified ClangOz compiler is utilised, both in serial, with all parallel flags disabled; and in parallel.

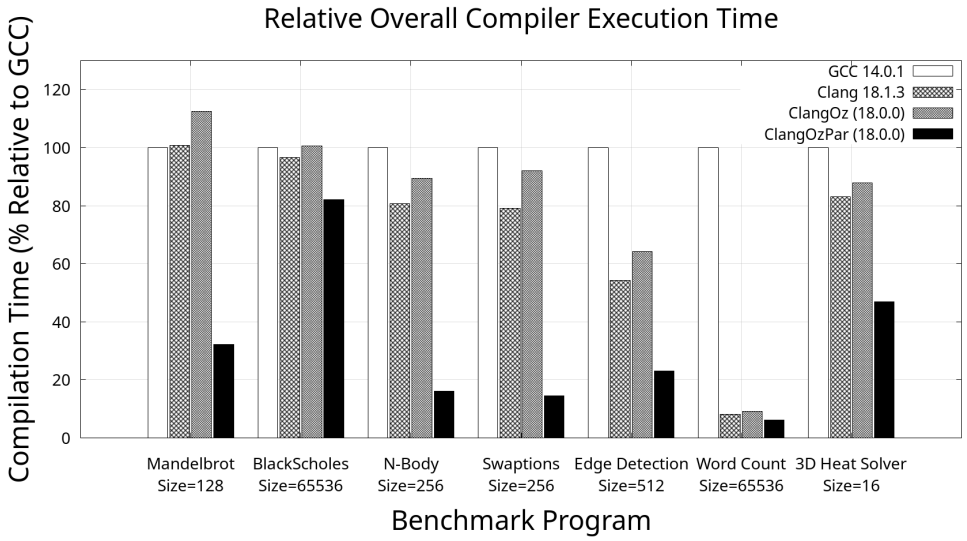

Figure 12: Overall black box compilation times of four distinct compiler configurations

The parallel invocation of ClangOz is configured with 8 threads. Each benchmark program and compiler invocation is executed thrice, with the average times presented. As our interest is in substantial metaprograms, the size of each benchmark should have been the largest already considered in this section; however GCC has a bug[5] which affects the Mandelbrot, Word

[4]Preparation for this experiment was simplified considerably alongside ClangOz's adoption of C++ standard execution policies.

[5]Separately, GCC uses a *signed* 32-bit integer to store its maximum number of *constexpr*

Counter and 3D Heat Solver benchmarks, compelling us to the smaller sizes of 128, 65536, and 16 for these three.

Figure 12 presents the results. Immediately apparent is the relatively low performance of GCC throughout. Only with the Mandelbrot benchmark does it find itself in the company of another compiler which is slower. Secondly, we observe that as ClangOz is built on an older version of Clang (18.0.0) than the Ubuntu package version, it is as expected that all benchmarks show its serial invocation performing relatively slowly in comparison to it. We can attribute this to the overhead of the sidelined ClangOz parallelisation apparatus; or perhaps a difference in build configuration. Finally, the parallel invocation of ClangOz is notable in producing lower (better) times than the other compiler configurations, for all benchmarks. The Word Count benchmark does *seem* to show near parity for the three Clang-based compilers, but recall that this is the *smallest* of the five sizes considered for this benchmark; GCC was only able to compile with 65536 words.

5.12. Mathematics Library Impact

A last experiment is introduced. The 3D Heat Solver benchmark involved a number of standard transcendental mathematical functions which were not available. Our own *constexpr* routines, based on glibc, provide only *exp*, *log* and *sqrt*. Meanwhile, implementations of proposal P1383 [9] for C++26 are not yet available. Consequently, we added code to support the GCE-Math library [10]; and not only to the 3D Heat Solver, but to all possible benchmarks under the control of an object-like macro: *USE_GCEM*. We then examine the performance implications of the mathematics library on five of the benchmark programs; excepting the 3D Heat Solver; and the Word Counter, as it makes no use of such mathematical functions.

Again, the parallel invocation of ClangOz is configured with 8 physical threads; and each benchmark program and compiler invocation is executed six times, with the average times presented. As GCC was not involved, we were able to use our largest benchmark size configurations; including 256 for the Mandelbrot program.

Figure 13 illustrates the dramatic impact of the choice of mathematics library upon compile-time evaluation times. While the use of the GCE Math library has little impact on performance for the BlackScholes, Swaptions and

operations; whereas Clang's use of a signed integer affords it double this quantity.

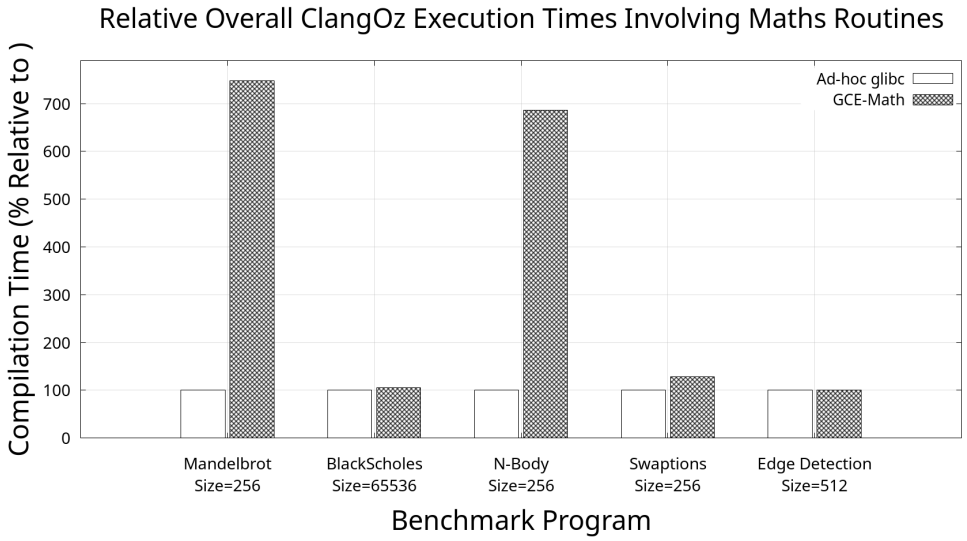

Figure 13: Overall black box ClangOz compilation times with differing mathematics libraries using 8 threads

SYCL Edge Detection benchmarks; the Mandelbrot and N-Body benchmarks show a considerable (∼700%) degradation in performance. The Mandelbrot and N-Body programs both make extensive, and exclusive, use of the square root function. The SYCL Edge Detection benchmark also uses only the square root function, though upon single-precision (32-bit) data, and with none of the iteration found in those two benchmarks. The BlackScholes and Swaptions benchmarks both utilise logarithmic, exponential, *and* square root functions.

6. Empirical Study via Github

Let us now consider the potential real-world impact of a compiler capable of utilising parallelism *today*. Do libraries already exist which include *constexpr* functions suitable for such acceleration? A first observation here is that the C++ standard library, provided with every C++ compiler, is the ideal target. While our own fork (C'est 2 [53]) of the GNU C++ library (libstdc++ [8]) is introduced in Section 4.1, both Microsoft's STL[6],

[6] `https://github.com/microsoft/STL`

and LLVM's libc++[7] could also easily adopt the required changes. As the C++ compiler associated with each of these is based on Clang (like ClangOz), integration issues are likely minimised; meanwhile, the code itself includes many *constexpr* functions implemented by suitable "map-reduce" algorithms. For example, within the Algorithms and Numerics library component, most non-*execution policy* overloads of function templates, including *std::for_each*, *std::transform_reduce*, and *std::inner_product* have been *constexpr* since C++20.

Looking more widely across the C++ development community, a number of well-known libraries concerned with compile-time programming can be considered. Enabled by C++11's *constexpr* functionality, Louis Dionne's Hana[8] library appeared in 2013, and pioneered an influential move away from conventional template metaprogramming syntax. Today, Hana is presented less often, with its last official release on Github in 2020. Parallelism could be introduced in Hana by replacing its ubiquitous use of parameter pack expansion, with calls to equivalent *constexpr* function templates from the C'est 2 library (such as *std::for_each*). More simply, *constexpr* Hana functions involving suitable for loops include *product_helper* and *compute_value*, as well as function templates within Hana's algorithm.hpp. As with Hana, the well-known Sprout library[9], updated 5 years ago, presents a modern *constexpr* API, with a design driven by C++14-era compile-time idioms; including recursion and pack expansions. Functions are named after those in the standard C++ Algorithms library, though no C++17 policy parameters have been introduced. So too, the Brigand metaprogramming library[10], last updated 2 years ago, utilises pack expansion extensively for mapping and iteration. Brigand's *count_bools* has a for loop which could easily be parallelised.

Recent C++ conferences continue to highlight presentations on relevant compile-time libraries and innovation. Joaquín M. López Muñoz discussed the topic of perfect hashing at using::cpp 2024. The associated Github repository[11] builds on its *constexpr_perfect_set* class, which contains a *constexpr* member function *construct* (used exclusively by both constructors)

[7] `https://libcxx.llvm.org`
[8] `https://github.com/boostorg/hana`
[9] `https://github.com/bolero-MURAKAMI/Sprout`
[10] `https://github.com/edouarda/brigand`
[11] `https://github.com/joaquintides/usingstdcpp2024`

which includes two substantial for loops, as well as calls to *std::sort* and *std::iota*. Alon Wolf presented on the venerable topic of compile-time parsing at C++Now 2024. Essentially all non-IO functions within the associated C++20 library[12] are *constexpr*, with the *ForEach* and *ForEachIndex* utility functions of interest here, though again built on pack expansion. At CppCon 2021, Ashley Roll presented on compile-time compression and resource generation, with three associated C++20 libraries[13], concerned respectively with compile-time string compression; cyclic redundancy checking (CRC) via compile-time look-up tables; and compile-time USB configuration descriptor generation. Each utilises *constexpr* or *consteval* function specifiers exhaustively, with code largely comparable to traditional runtime code: involving parallelisable for loops, and a range of standard C++ Algorithm library function calls over containers. Strongly typed handling of units and quantities, and dimensional analysis, is often touted for future ISO C++ standardisation. On this topic, Mateusz Pusz presented the mp-units[14] library at CppOnline 2024. The majority of the functions in this library are qualified either as *constexpr* or *consteval*; with algorithms involving suitable for loops, as well as an algorithm.h header containing analogues of functions from the C++ standard Algorithm library.

6.1. Systematic Repository Search

While Section 6 considers relevant compile-time libraries which have been presented internationally, one may still be left curious as to the impact of this programming idiom within the wider C++ library ecosystem. For many, the *constexpr* function specifier has become a default; even in code assumed to target runtime contexts only, its semantics (which subsumes the older *inline* specifier) are often useful in high-performance contexts. Of the approximately 384 proposals listed by Cppreference[15] as *adopted* by the ISO C++ standard in recent years (from C++20 to C++26), 34 ($\approx 9\%$) include the *constexpr*[16] keyword *in their title*.

[12] `https://github.com/a10nw01f/Gen`

[13] `https://github.com/AshleyRoll/squeeze`, `https://github.com/AshleyRoll/crc_cpp` and `https://github.com/AshleyRoll/cpp_usbdescriptor`

[14] `https://github.com/mpusz/mp-units`

[15] `https://en.cppreference.com/w/cpp/compiler_support`

[16] Of course this omits many relevant proposals, including in C++26: "User-generated `static_assert` messages" (P2741R3); and "<linalg>: A free function linear algebra inter-

To examine the context historically we have executed a systematic review of annual C++ *constexpr*-oriented compile-time library creation on Github.com, starting from 2011; facilitated by the command-line interface to GitHub: Github CLI. As well as the "constexpr" search term, we separately reviewed results from "metaprogramming" and "reflection"; and included default fields for each query: *name*, *description* and *topic*. Such searches are within repositories identified as "C++" within the separate *language* category. An author may not have categorised their repo as "C++", so while adding the C++ language constraint to the "constexpr" search will needlessly (as *constexpr* is a C++-only term) reduce the result count, retaining it allows for a more direct comparison against other search terms; "metaprogramming" or "reflection" could easily be associated with other languages. Also included are results from repositories categorised as "C++", with *no* further search constraints. Quantities graphed here are measured via a larger y-axis scale shown on the right. Repository forks are not included, as they may include vestigial pull request artifacts. We lastly note that reflection can refer to either runtime or compile-time (or both) varieties; and that the "metaprogramming" spelling was selected over "meta-programming" as it has around 3x the search results. Results are depicted in Figure 14.

Figure 14 shows a steadying in the rate of C++ *constexpr* repository creation since 2018; following growth since the keyword's inception in 2011. Also, each year following the release of revisions to the C++ standard coincides with a modest surge in repository creation; at least after C++11, C++14, and C++17. Surprisingly, the C++20 edition does not have this accolade. This is against a setting of strong, consistent growth in the numbers of C++ repositories; with around 200,000 *more* repositories created in 2023 compared to 2022.

7. Conclusion

As the C++ standard has evolved, additional compile-time language features have been added, extending the reach of compile-time metaprogramming. As C++'s compile-time repertoire and use has expanded, the problem of increasing compilation times becomes prominent, leading to adverse effects on programmer work flow. This opens up the question of how to alleviate

face based on the BLAS" (P1673R13), which introduces a suite of consistently *constexpr* functions; many with an execution policy parameter to support parallel execution.

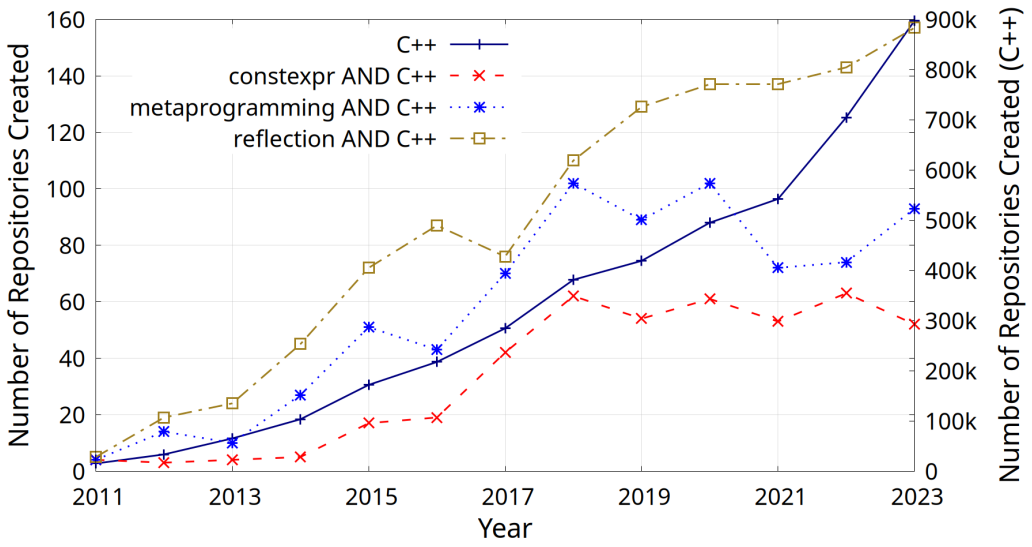

Figure 14: Number of Github repositories created annually, and labelled as "C++" (right y-axis); alongside those also yielding a successful match with search terms: "constexpr", "metaprogramming", and "reflection" (left y-axis).

the issue. In the project introduced here, the option of acceleration through multi-threaded data-parallelism, within the compiler is investigated. ClangOz, an extended Clang compiler for C++ is introduced that can parallelise *for* loops during constant evaluation; including with reduction/accumulation. Therein, intrinsic functions allow users to explicitly relay information to the compiler about the loop being parallelised. This firstly allows users the flexibility to implement their own low-level compile-time parallel algorithms, while understanding the intrinsics' semantics. Together, the compiler and intrinsics create a framework for accelerating constant expression evaluation.

This low-level functionality has been utilised to provide a *high-level API*, which builds on recent C++ standard library support for parallelism to implement 30 *constexpr* parallel function templates. These functions are based on existing function template signatures within the C++ standard library, and differ only in that calling them with a standard *std::execution::par* policy parameter object as a first argument provokes parallel constant expression evaluation through C++ overloading. Seven compile-time benchmarks were implemented that utilise a *constexpr* parallel *std::for_each* or *std::transform_reduce* from this extended library. Through testing of these

benchmarks it was shown that the ClangOz framework can have large performance benefits; with around 86% of ideal speedup (i.e. *efficiency*) on the Swaptions benchmark when configured with one thread per physical core, and performance degradation in only one instance (i3D Heat Solver). These benchmarks also show that the complexity of the framework can be hidden within a library; shielding users from the onus of understanding low-level compiler intrinsics, while maintaining high performance. Benchmark results nevertheless indicate that there is still room for improvement.

The current parallelisation process has some areas that could be addressed to improve performance. One issue stems from the fact that the data copying process required when forking and joining threads can be expensive. This leads to significant startup costs, meaning that multiple sequential parallel regions for trivial amounts of computation are slower than if done sequentially. Large data dependencies can also have an impact on how much of a performance increase can be obtained from the parallelisation process. Optimising or removing the need for the cloning process would likely improve performance. Another issue is the lack of workload balancing in the implementation, which necessitates that users must choose their thread partitioning carefully. When data is indivisible by the thread count this can have a performance penalty as one thread will keep the others waiting as it deals with the excess data. Adding a workload balancing component within the compiler could improve the parallelisation algorithm's overall performance. Recent work [60] on Clang's constant expression evaluator by Timm Baeder of Red Hat also holds promise, both in terms of the potential for serial performance improvements; as well as in the software engineering advantages of an industry-supported, modular architecture.

Finally, our study involving Github in Section 6 leads us to conclude that there are multiple community projects with potential to incorporate Clangoz's parallelisation features. But while *constexpr* within the C++ standard library, and vendor support of it, remains young, minor refactorings, away from today's prevailing use of template parameter packs for compile-time iteration and mapping, should be expected. In their place, developers can apply either ClangOz's low-level intrinsics API; or a selection of high-level *constexpr*-annotated standard Algorithm/Numerics library function templates would be required, as provided by C'est 2.

Acknowledgements

The authors wish to thank the Royal Society of Edinburgh for their support through the Saltire International Collaboration Award (Grant Number 1981).

Appendix A. Intrinsic Functions Table

Table A.1: The ClangOz Intrinsic Functions

```
template <class T, class U>
constexpr void __BeginEndIteratorPair(T& Begin, U& End);
```

Indicates the range of a *for* loop, allowing the partitioning process to split work across multiple threads. The *__BeginEndIteratorPair* or *__PartitionUsingIndex* intrinsic are required and the minimum necessary for parallelisation.

- **Begin**, **End**: Indicates the beginning and end of the loop's range.
- **Type requirements**: T and U must be pointers; or single member iterators where the member is a pointer.

```
template <class T, class U>
constexpr void __PartitionUsingIndex(T LHS, U RHS, RelationalType RelTy);
```

Indicates the range of a *for* loop, allowing the partitioning process to split work across multiple threads. The *__BeginEndIteratorPair* or *__PartitionUsingIndex* intrinsic are required and the minimum necessary for parallelisation.

- **LHS**, **RHS**: Indicates the beginning and end of the loop's range.
- **RelTy**: Indicates the relational operator used within the loop's condition e.g. $>, <, !=, <=, >=$.
- **Type requirements**: T and U must be a numeric type, such as an integer.

```
template <class T>
constexpr void
__IteratorLoopStep(T& StartIter, OperatorType OpTy, const T& BoundIter);
```

States *StartIter* is bound to the loops step. Thread clones will initially be offset by invoking operator based mutation the same number of steps taken by the thread partitions loop at its start point. The operator used for mutation is indicated by *OpTy*.

- **StartIter**: Indicates the variable that will be offset.
- **OpTy**: Indicates the prefix or postfix operator (e.g. ++) used for mutation.
- **BoundIter**: Indicates the boundary of *StartIter* if one exists, preventing offsetting past the boundary.
- **Type requirements**: *T* must be a pointer; or a single member iterator where the member is a pointer.

```
template <class T>
constexpr void
__ReduceVariable(T Var, ReductionType RedTy, OperatorType OpTy);
```

Indicates that a container or value should be reduced when the launched threads are joined. Three types of reduction are supported *PartitionedOrderedAssign*, *OrderedAssign* or *Accumulate*.

- **Var**: Notates the variable that should be reduced on thread completion.
- **RedTy**: Indicates the reduction method to be used by the compiler.
- **OpTy**: States the operator, if any, used to mutate the variable in the reduction step.
- **Type requirements**: T must be a vector or array iterator or numeric value.

Appendix B. Code Example from Section 4.1

```
#include <algorithm>
#include <deque>
#include <forward_list>
#include <functional>
#include <iostream>
#include <list>
#include <map>
#include <memory>
#include <numeric>
#include <set>
#include <sstream>
#include <string>

constexpr bool doit()
{
  using namespace std;

  list<int> l{1, 2, 3};
  forward_list<int> fl{2, 3, 4};
  deque<int> dq(fl.begin(), fl.end());
  set<int> s;

  set_intersection(dq.begin(), dq.end(), l.begin(), l.end(),
                   inserter(s, s.end()));
  function<int()> f = [&] { return accumulate(s.begin(), s.end(), 0); };
  auto x = f();

  map<string, int> m{{"five", 5}, {"World ", 6}};
  auto p = make_shared<int>(m["five"]);

  istringstream iss("Hello ");
  cout << iss.str() << begin(m)->first << *p << ' ' << x << endl;

  return *p == x;
}

int main(int argc, char *argv[])
{
  static_assert(doit());
  return doit() ? 0 : 1;
}
```

References

[1] Todd, L. Veldhuizeni, C++ Templates are Turing Complete, Available at citeseer.ist.psu.edu/581150.html (2003).

[2] D. R. Gabriel, B. Stroustrup, J. Maurer, Generalized constant expressions–revision 5, Tech. rep., ISO SC22 WG21 TR (2007).

[3] ISO/IEC JTC 1/SC 22, ISO/IEC 14882:2011 Programming languages — C++.

[4] A. Gozillon, ClangOz (2020).
URL `https://github.com/agozillon/ClangOz`

[5] C. Lattner, LLVM and Clang: Next generation compiler technology, in: The BSD conference, Vol. 5, 2008.

[6] A. Gozillon, H. Haeri, J. Riordan, P. Keir, Compiler Support for Parallel Evaluation of C++ Constant Expressions, in: M. Ganzha, L. Maciaszek, M. Paprzycki, D. Ślęzak (Eds.), Proceedings of the 18th Conference on Computer Science and Intelligence Systems, Vol. 35 of Annals of Computer Science and Information Systems, IEEE, 2023, p. 481–490. `doi:10.15439/2023F4268`.

[7] A. Gozillon, S. H. Haeri, J. Riordan, P. Keir, Performance analysis of compiler support for parallel evaluation of c++ constant expressions, in: A. Jarzębowicz, I. Luković, A. Przybyłek, M. Staroń, M. O. Ahmad, M. Ochodek (Eds.), Software, System, and Service Engineering, Springer Nature Switzerland, Cham, 2024, pp. 129–152. `doi:10.1007/978-3-031-51075-5\_6`.

[8] The Free Software Foundation (FSF), The GNU C++ Library (Feb. 2008).
URL `https://gcc.gnu.org/onlinedocs/libstdc++`

[9] O. J. Rosten, More constexpr for cmath and complex (Jun. 2023).
URL `https://wg21.link/P3037R1`

[10] K. O'Hara, GCE-Math (2017).
URL `https://github.com/kthohr/gcem`

[11] The Free Software Foundation (FSF), The GNU C Library (Jan. 1988).
URL https://www.gnu.org/software/libc

[12] H. Alblas, R. op den Akker, P. O. Luttighuis, K. Sikkel, A bibliography on parallel parsing, ACM Sigplan Notices 29 (1) (1994) 54–65. doi:10.1145/181577.181586.

[13] H. P. Katseff, Using data partitioning to implement a parallel assembler, in: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems, 1988, pp. 66–76. doi:10.1145/62115.62123.

[14] V. Seshadri, S. Weber, D. Wortman, C. Yu, I. Small, Semantic analysis in a concurrent compiler, in: Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation, 1988, pp. 233–240. doi:10.1145/53990.54013.

[15] G. U. Srikanth, Parallel lexical analyzer on the cell processor, in: 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion, IEEE, 2010, pp. 28–29. doi:10.1109/SSIRI-C.2010.16.

[16] T. Gross, A. Sobel, M. Zolg, Parallel compilation for a parallel machine, in: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, 1989, pp. 91–100. doi:10.1145/73141.74826.

[17] ISO/IEC JTC 1/SC 22, ISO/IEC 14882:2020 Programming languages — C++.

[18] P. Dimov, L. Dionne, N. Ranns, R. Smith, D. Vandevoorde, More constexpr containers (Jul. 2019).
URL https://wg21.link/P0784R7

[19] A. Sutton, C++ extensions for Concepts (Jul. 2017).
URL https://wg21.link/P0734R0

[20] H. Sutter, Metaclasses: Generative C++ (Feb. 2018).
URL https://wg21.link/P0707R3

[21] W. Childers, P. Dimov, D. Katz, B. Revzin, A. Sutton, F. Vali, D. Vandevoorde, Reflection for C++26 (Jun. 2024).
URL https://wg21.link/P2996R4

[22] H. Dusíková, Compile time regular expressions (2016).
URL https://github.com/hanickadot/compile-time-regular-expressions

[23] M. Sánchez, tinyrefl (2018).
URL https://github.com/Manu343726/tinyrefl

[24] N. J. Bouman, Multiprecision Arithmetic for Cryptology in C++ - Compile-Time Computations and Beating the Performance of Hand-Optimized Assembly at Run-Time (2018). doi:10.48550/arXiv.1804.07236.

[25] B. Fahller, lift (2017).
URL https://github.com/rollbear/lift

[26] G. Steele, Common LISP: the language, Elsevier, 1990.

[27] A. Alexandrescu, The D programming language, Addison-Wesley Professional, 2010.

[28] N. D. Matsakis, F. S. Klock, The rust language, ACM SIGAda Ada Letters 34 (3) (2014) 103–104. doi:10.1145/2692956.2663188.

[29] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah, Julia: A fresh approach to numerical computing, SIAM review 59 (1) (2017) 65–98. doi:10.1137/141000671.

[30] C. McCord, Metaprogramming Elixir, 1st Edition, Pragmatic Bookshelf, 2015.

[31] S. Baxter, circle (2019).
URL https://github.com/seanbaxter/circle

[32] N. D. Jones, An introduction to partial evaluation, ACM Computing Surveys (CSUR) 28 (3) (1996) 480–503.

[33] T. L. Veldhuizen, C++ templates as partial evaluation, arXiv preprint cs/9810010 (1998).

[34] A. Tyurin, D. Berezun, S. Grigorev, Optimizing gpu programs by partial evaluation, in: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2020, pp. 431–432.

[35] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, B. Schmidt, Anydsl: A partial evaluation framework for programming high-performance libraries, Proceedings of the ACM on Programming Languages 2 (OOPSLA) (2018) 1–30.

[36] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, M. Grimmer, Practical partial evaluation for high-performance dynamic language runtimes, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017, pp. 662–676.

[37] C. Consel, O. Danvy, Partial evaluation in parallel, Lisp and Symbolic Computation 5 (4) (1993) 327–342.

[38] M. Sperber, P. Thiemann, H. Klaeren, Distributed partial evaluation, in: Proceedings of the second international symposium on Parallel symbolic computation, 1997, pp. 80–87.

[39] A. Bouter, T. Alderliesten, A. Bel, C. Witteveen, P. A. Bosman, Large-scale parallelization of partial evaluations in evolutionary algorithms for real-world problems, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2018, pp. 1199–1206.

[40] K. Kennedy, J. R. Allen, Optimizing compilers for modern architectures: a dependence-based approach, Morgan Kaufmann Publishers Inc., 2001.

[41] ISO/IEC JTC 1/SC 22, Technical Specification for C++ Extensions for Parallelism (2018).

[42] L. Dagum, R. Menon, Openmp: an industry standard api for shared-memory programming, IEEE computational science and engineering 5 (1) (1998) 46–55. doi:10.1109/99.660313.

[43] B. Revzin, Relaxing some constexpr restrictions (Jan. 2022).
URL https://wg21.link/P2448R2

[44] B. Revzin, Permitting static constexpr variables in constexpr functions (Nov. 2022).
URL https://wg21.link/P2647R1

[45] V. Voutilainen, Non-literal variables (and labels and gotos) in constexpr functions (Jul. 2021).
URL https://wg21.link/P2242R3

[46] A. Fertig, Making std::unique_ptr constexpr (Nov. 2021).
URL https://wg21.link/P2273R3

[47] D. Goncharov, A more constexpr bitset (Jun. 2022).
URL https://wg21.link/P2417R2

[48] B. Revzin, Missing constexpr in std::optional and std::variant (Feb. 2021).
URL https://wg21.link/P2231R1

[49] P. Dimov, Making std::type_info::operator== constexpr (May 2021).
URL https://wg21.link/P1328R1

[50] D. Goncharov, A. Karaev, Add Constexpr Modifiers to Functions to_chars and from_chars for Integral Types in <charconv> Header (Sep. 2021).
URL https://wg21.link/P2291R3

[51] E. J. Rosten, O. J. Rosten, constexpr for <cmath> and <cstdlib> (Nov. 2021).
URL https://wg21.link/P0533R9

[52] P. Keir, C'est (Apr. 2020).
URL https://github.com/SCT4SP/cest

[53] P. Keir, C'est 2 (Feb. 2023).
URL https://github.com/SCT4SP/gcc

[54] C. Bienia, S. Kumar, J. P. Singh, K. Li, The PARSEC benchmark suite: Characterization and architectural implications, in: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, 2008, pp. 72–81. doi:10.1145/1454115.1454128.

[55] D. Bagley, The computer language benchmarks game (Apr. 2001).
URL https://benchmarksgame-team.pages.debian.net/benchmarksgame

[56] The Khronos Group, SYCL 1.2.1 Specification (Nov. 2019).
URL https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

[57] B. A. Lelbach, The C++17 Parallel Algorithms Library and Beyond, CppCon 2016 (2016).
URL https://github.com/CppCon/CppCon2016

[58] Y. Asahi, T. Padioleau, G. Latu, J. Bigot, V. Grandgirard, K. Obrejan, Performance portable Vlasov code with C++ parallel algorithm, in: 2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2022, pp. 68–80. doi:10.1109/P3HPC56579.2022.00012.

[59] J. Penuchot, J. Falcou, ctbench - compile-time benchmarking and analysis, Journal of Open Source Software 8 (88) (2023) 5165. doi:10.21105/joss.05165.

[60] T. Baeder, A new constant expression interpreter for Clang, Part 2 (2023).