Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 481–490
DOI: 10.15439/2023F4268
ISSN 2300-5963 ACSIS, Vol. 35

Compiler Support for Parallel Evaluation of C++ Constant Expressions

Andrew Gozillon
0000-0001-7558-7166
Advanced Micro Devices AB,
Nordenskioldsgatan 11 A, Office 233
211 19 Malmö, Sweden
Email: andrew.gozillon@amd.com

Seyed H. HAERI
0000-0002-7969-8573
University of Bergen, Norway & PLWorkz R&D, Belgium
Av. Chapelle-aux-Champs 49, Brussels, Belgium
Email: hossein@uib.no

James Riordan, Paul Keir
0000-0001-6516-9446
0000-0002-4781-9377
University of the West of Scotland
High St., Paisley PA1 2BE, Scotland, United Kingdom
Email: {james.riordan, paul.keir}@uws.ac.uk

***Abstract*—Metaprogramming, the practice of writing programs that manipulate other programs at compile-time, continues to impact software development; enabling new approaches to optimisation, static analysis, and reflection. Nevertheless, a significant challenge associated with advanced metaprogramming techniques, including the *constexpr* functionality introduced to C++ in 2011, is an increase in compilation times. This paper presents ClangOz, a novel Clang-based research compiler that addresses this issue by evaluating relevant constant expressions in parallel, thereby reducing compilation time.**

ClangOz includes a set of compiler intrinsics that allows developers to parallelise their own code and take full advantage of recent constexpr language support. By utilising parallel evaluation, ClangOz significantly reduces the compile time for metaprogramming-intensive codebases, enhancing developer productivity and iterative software development processes.

Benchmark results demonstrate the performance advantage of ClangOz over traditional compilers, including a decrease in compilation times across all benchmarks; and parallel efficiency of up to 95% in one case. The evaluation of constant expressions in parallel unlocks substantial speedups, enabling developers to leverage advanced metaprogramming techniques without sacrificing compilation efficiency.

We highlight the opportunities afforded by the constexpr functionality and emphasise the importance of compiler support for efficient metaprogramming. By introducing ClangOz, a compiler tailored for parallel evaluation of relevant constant expressions, developers can utilise modern metaprogramming while minimising compilation times parametrically.

I. 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 constant expressions to the language; a concept proposed in 2003 [2]; and added in C++11 with the inclusion of the *constexpr* specifier. A constant expression is an expression which would remain constant at runtime and could thus be evaluated at compile time. The *constexpr* specifier allows functions and variable declarations to assert that they can be evaluated at compile time. 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 evaluated at compile time. However, as more components are evaluated at compile time, so too do compilation 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.

In this article we introduce ClangOz [3], an experimental Clang [4] compiler which adds support for the parallel execution of for-loops at compile time. ClangOz seeks to give users control of parallelism through compiler intrinsics. The 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.

A survey of related and relevant literature is presented in Section II. Section III covers the ClangOz compiler, discussing its architecture; parallel constant expression evaluation; and intrinsics library, along with a concise implementation example of *std::for_each*. Before concluding in Section V, Section IV reports on experiments, with benchmarks implemented using the novel compile time parallelism feature, and considers performance and scaling in comparison with serial counterparts.

II. 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 [5], assembling [6], semantic analysis [7], lexical analysis [8] and code generation [9] 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 [10]) dynamic memory allocation and deallocation at compile time was added [11]. This allowed the creation of variable size containers at compile time. C++20 also introduced a feature called *Concepts* [12], allowing 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 [13] and reflection [14]. 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.

Additions like these to the language specification have allowed for projects that were previously impossible. The processing of regular expressions at compile time [15], static reflection through a library rather than the language [16], big-integer computation [17] and compile time functional composition [18] are prominent examples. Such projects require a sizable amount of computation at compile time, and would benefit from acceleration by *constexpr* parallelisation.

Providing language features to allow processing at compile time is not unique to C++. Lisp [19], D [20], Rust [21], Julia [22], Elixir [23] and Circle [24] 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 intent to simplify 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 at compile time has parallels in the field of partial evaluation [25], where programs are specialised dynamically at runtime or statically at compile time 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 *constexpr* within C++ to specialise code; and we recognise that the comparison of partial evaluation and C++'s compile time features has been made before [26]. 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 [27]; creating domain specific languages utilising partial evaluation to facilitate high-performance libraries for accelerators [28]; and in the development of compilers and interpreters for dynamic languages that utilise partial evaluation to speculatively optimise code [29]. 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 [30]; distributed parallelisation of partial evaluators within programming languages [31]; and parallelisation of partial evaluations within evolutionary algorithms [32].

III. 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 [33].

When the following constraints have been met within a *constexpr* function, ClangOz will use these intrinsics to gather the information required to partition the loop body across multiple CPU threads. There are two constraints that have to be met by a *for* loop to be *constexpr* parallelised. First, it must be within a *constexpr* function; and adjacent to the appropriate parallelisation intrinsics. Second, the enclosing function must include our new C++ execution policy class [34] within its parameter list; and be invoked with the corresponding object as an argument. An execution policy parameter allows an algorithm designer to overload a function's behaviour based on the

[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.

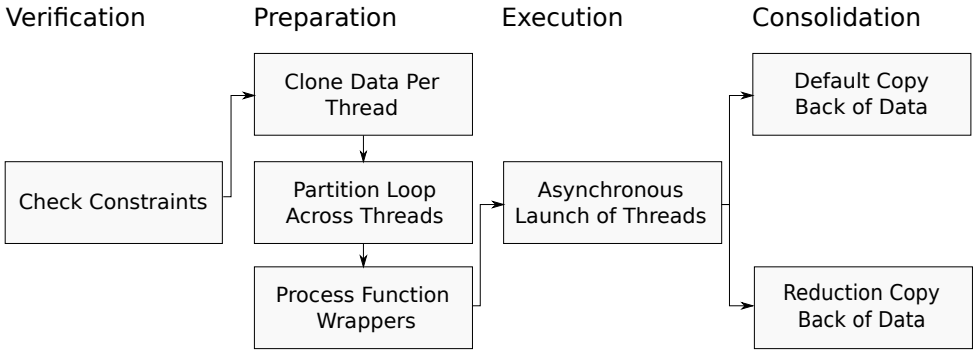

Fig. 1. Compilation Phases involved in the Parallelisation Process

type of each distinct policy. For example, a *std::for_each* function may be passed a *std::execution::parallel_policy* which indicates that the *std::for_each* should select an overload that has a parallelised implementation. In the case of ClangOz a new *constexpr_parallel_policy* was added to ClangOz's C++ standard library (a modified version of Clang's implementation of the C++ standard library: libc++) which is used to indicate that a function should be *constexpr* parallelised if possible.

These constraints define a minimal C++ API that *constexpr* functions must meet to undergo the parallelisation process. The constraints were added as a way to limit the scope at which ClangOz would try to apply its parallelisation process.

There are some limitations of the ClangOz compiler worth noting. First, there is no support for nested parallelism; only the outer loop of a loop nest is parallelisable. Second, only a single loop is parallelisable within each function. Thirdly, only container argument types owning *contiguous* data are supported; and container objects must utilise a pointer-based iterator. An example usage of a *constexpr* parallel *std::for_each* from ClangOz's modified libc++ can be found in Listing 1. *execution::ce_par* is a simple, 1-byte, pre-constructed *constexpr_parallel_policy* object. Passing this policy into the *std::for_each* indicates to use a parallel implementation of the function; if one exists. The 4th argument, a C++ lambda function, is then executed in parallel on the elements of the *std::array*.

Listing 1 ClangOz's modified libc++ supports a new *ce_par* policy parameter, allowing users to avoid compiler intrinsics.

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

The parallelisation process takes place within Clang's constant expression evaluator. The constant expression evaluation executes within the frontend, usually during the semantic analysis process; 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 at compile time 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.

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*.

A. The Parallelisation Process

The parallelisation process consists of four phases (See Fig. 1). The first phase is *verification*, confirming that the two constraints have been satisfied. These constraints are checked whenever a *for* loop is encountered within a *constexpr* context.

TABLE I
THE CLANGOZ INTRINSIC FUNCTIONS

Intrinsic Function Code	Function and Parameters Synopsis

`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 loops range.
	• **Type requirements**: *T* and *U* must be a one member iterator where the member is a pointer or 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 loops 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 one member iterator where the member is a pointer or 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.

The first step checks the enclosing context is a function, before iterating over the function's parameters to detect if the function takes a *constexpr_parallel_policy* as an argument. As nested parallelism is not supported, the second check makes sure that no other *constexpr* parallel tasks are in flight.

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 III-B.

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

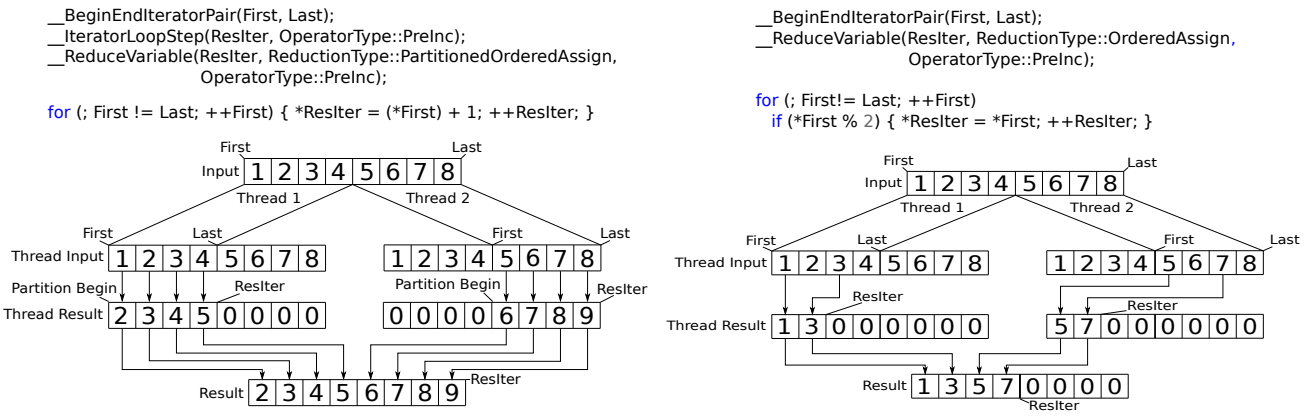

Fig. 2. Example PartitionedOrderedAssign (Left) and OrderedAssign (Right) Reduction With Two Threads

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 to 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, and then each pointer can be 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 III-B. 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 statements. 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 III-B.

B. The Intrinsic Functions

The compiler intrinsics are used as standard C++ functions, to 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

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

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

Fig. 3. *Constexpr* parallelised libc++ *std::for_each* implementation

with OpenMP [35] 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 I. The intrinsics have no body and are no-ops at runtime with a trivial overhead at compile time. 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 allow 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 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 in Fig. 2. 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* (also Fig. 2) 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.

C. An Example Constexpr Parallel Function

Within ClangOz's libc++ library, 30 of the functions contained inside the Algorithms and Numerics libraries have been *constexpr* parallelised. In Fig. 3 a *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.

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.

IV. Parallelism Benchmarking

Three *constexpr* programs based on existing benchmarks were created to test the performance of the *constexpr* parallelism implementation. The original benchmarks were modified to utilise function templates from ClangOz's C++ standard library supporting *constexpr* parallel execution. Benchmark selection was also mindful that certain language features are not available at compile time; for example assembly instructions or *goto* statements.

The *Blackscholes* benchmark is taken from The Princeton Application Repository for Shared-Memory Computers (PARSEC) [36]. The PARSEC suite contains several multi-threaded programs that explore different workloads on shared memory

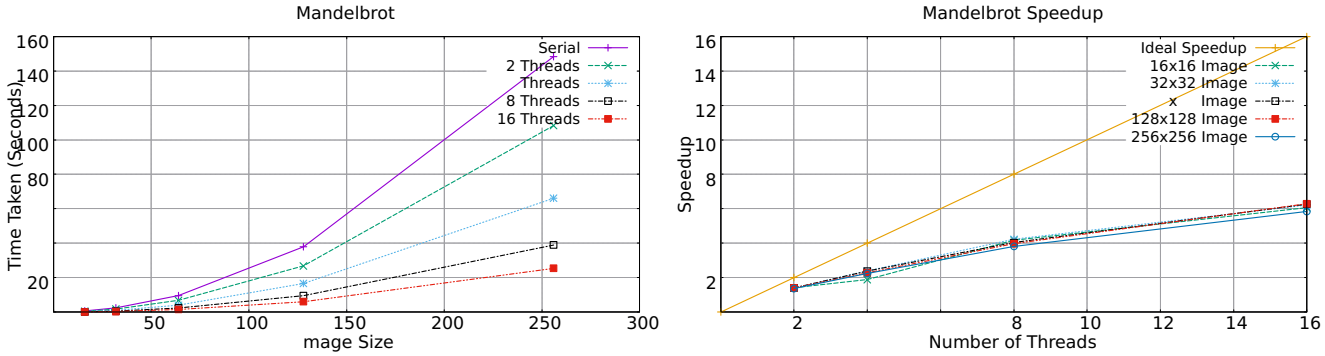

Fig. 4. Compilation Time and Speedup for the Mandelbrot Benchmark

architectures. *Blackscholes* processes financial data using a partial differential equation.

The *N-Body Problem* and *Mandelbrot* benchmarks are based on solutions provided to The Computer Language Benchmarks Game [37]. These are originally micro-benchmarks with the goal of testing performance of different programming languages as opposed to directly testing parallel performance.

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 a loop's range. The parallelised loop regions are indicated by an invocation of a *std::for_each*, which has been adapted to support *constexpr* parallelisation. The *std::for_each* invokes a function on each piece of data, in this case each thread will be given a set of data to invoke the function on individually.

The performance data gathered for each benchmark is displayed using two types of graph. The first is a line graph comparing serial and parallelised execution times (during compilation). Each of the plots in these graphs is calculated by averaging six separate runs of each benchmark and data size configuration. The second type of graph displays *parallel speedup*; comparing times when using different numbers of threads against the ideal speedup on different problem sizes for the benchmark. The ideal speedup is a one to one match for the number of threads used. A cumulative speedup graph is also provided, including speedups for all of the benchmarks.

A. Timing Compile Time Performance

Performance of the *constexpr* paralleliser is measured by comparing the speed of the parallelised constant expression evaluator against the original serial implementation on each of the benchmarks. The benchmarks are tested with different numbers of threads using an Intel Core i9-12900K CPU, containing eight performance cores, and with support for 16 hardware threads via hyper-threading. The benchmarks were run under the WSL2 virtual machine on Windows 11; and executed using two, four, eight and sixteen threads.

Time is measured from the beginning of a parallel region to its end, rather than timing the length of the entire program; this is to allow us to focus on the regions of interest. The same location is measured for the serial execution. Three timing intrinsics were implemented to allow measurement of the phase within the ClangOz compiler.

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 time measurement 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 III-A must be processed, which also adds an extra performance cost when evaluating the original serial implementation.

B. Mandelbrot

The Mandelbrot benchmark consists of three main areas of computation that are executed in parallel using *std::for_each*. Two initialisation steps populate a 2D array of complex values (a class containing two 64-bit floats) per pixel. 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 Fig. 4 show that the increase in data size gradually progresses towards higher polynomial growth as the number of threads diminish. With more threads, the increase in data size has less impact on compilation time. Breaking the computation into two separate parallel regions, requiring two thread group launches, has had minimal impact on this benchmark. The speedup graph shows that across all image sizes the performance improvement is similar.

C. Blackscholes

Blackscholes has two areas of computation which are parallelised, requiring two separate calls to *std::for_each*. The first is the main computation which computes the Black-Scholes equation; the second verifies the results from the

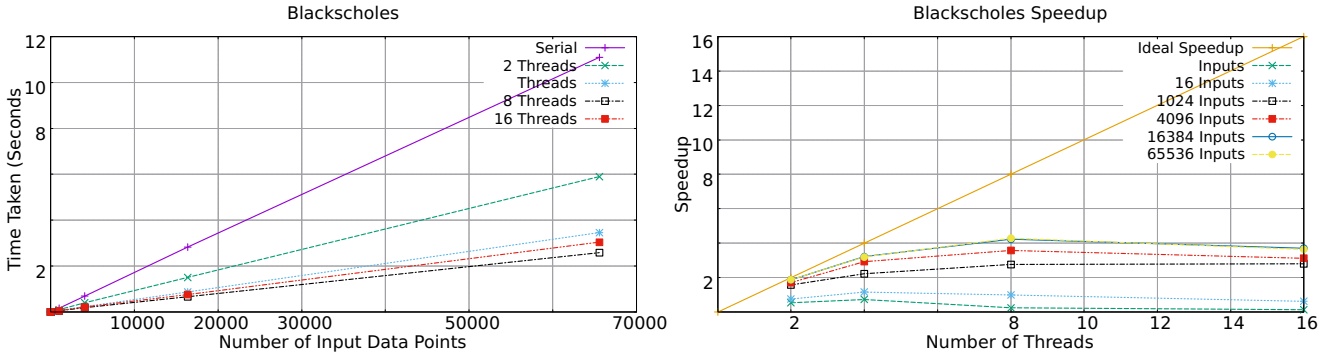

Fig. 5. Compilation Time and Speedup for the Blackscholes Benchmark

first computation. The main data that requires cloning in this benchmark is a 1D array containing a structure for each input that owns 9 literal values.

The Blackscholes data in Fig. 5 shows promising performance increases when utilising both two and four threads. The speedup graph indicates that the highest speedup occurs when larger input sets are used. Smaller data sets still yield an increase but plateau or fall slightly at four threads. The poor performance of the 4 and 16 sized data sets can be accounted for by the increased cost of preparing and launching threads outweighing the work required to process these tiny data sets.

D. 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 launches of threads occur. To allow a range of body numbers, the number of iterations is restricted to 32; while still avoiding compilation limits. The main data cloned within the benchmark are the bodies; structures containing seven 64-bit floats stored contiguously within an array. The number of bodies is the parameter that is varied within this benchmark.

The graphs in Fig. 6 show that increasing the number of threads again outperforms the serial implementation, however using two 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 (which utilises hyper-threading).

E. Benchmark Comparison

The speedup graph in Fig. 7 compares the most time consuming variations of each benchmark against each other and indicates which benchmarks achieved the best performance after parallelisation. The N-Body benchmark is the best performer reaching the closest to the ideal speedup across all of the benchmarks. This is due to the trivial size and simplicity of the data that requires cloning. Until eight threads, the Mandelbrot benchmark is the furthest from the ideal speedup; possibly according to the number of parallel regions, compared to the other benchmarks. Two of the parallel regions execute relatively small computations, to fill small arrays of data, while cloning a significant amount within the context of each parallel function call. The possibility of multiple thread launches causing cloning to have a negative impact is highlighted by Blackscholes performing better than Mandelbrot despite having a similar amount of data to clone per parallel region, but less parallel regions overall. With sixteen threads, hyperthreading should allow two threads to run efficiently on each of the eight performance cores, but only the Mandelbrot benchmark shows a reduction in execution time at the largest thread count.

The results indicate that the parallelisation of *for* loops during compile time evaluation can lead to notable speedups when a large portion of the program conforms reasonably to a loop. However, the cost of cloning and partitioning 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, containers like *std::array* which have no conflicting data accesses across threads should not require cloning. This could yield a performance increase in most of these benchmarks as the main input data is generally contained in an array. Data that is not required for the execution of the parallel *constexpr* function could also be elided from the 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.

V. 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 its use has expanded, the problem of increasing

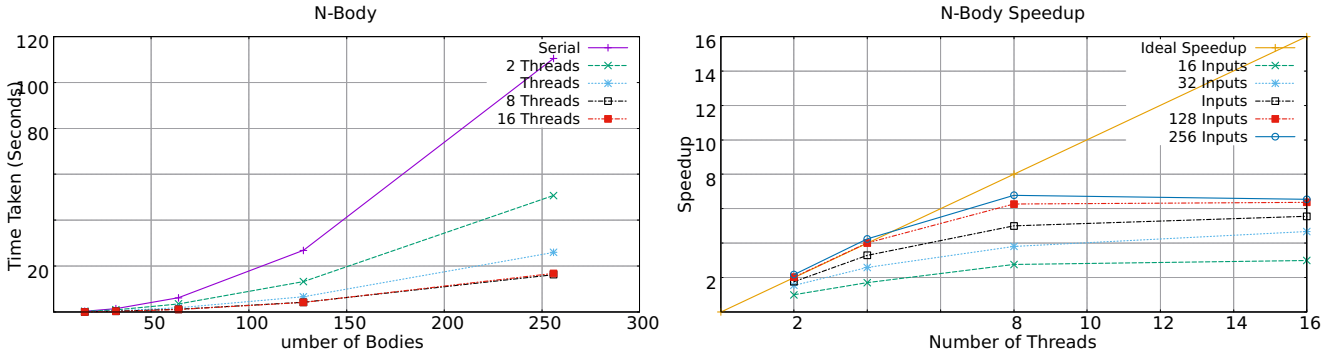

Fig. 6. Compilation Time and Speedup for the N-Body Benchmark

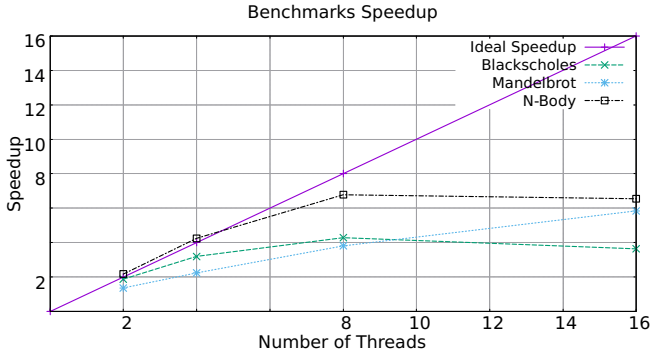

Fig. 7. Speedup Graph Comparison of all Benchmarks Compilation Time

compilation times becomes prominent, leading to adverse effects on programmer work flow. This opens up the question of how to alleviate 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 at compile time; 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 then 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 by a single argument; the policy parameter, providing access to parallelism through C++ overloading. Three compile time benchmarks were implemented that utilise a *constexpr* parallel *std::for_each* from this extended library. Through testing of these benchmarks it was shown that the ClangOz framework can have large performance benefits; with up to 95% parallel efficiency on one benchmark, and above 50% on average. These benchmarks also show that the complexity of the framework can be hidden within a library; removing users from the onus of understanding low-level 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 you will get from the parallelisation process. Optimising or removing the need for the cloning process would likely improve performance. Another issue is the lack of work load 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. A solution would be implementing a work load balancing algorithm within the compiler. These adjustments to the compiler could improve the parallelisation algorithm's overall performance.

ACKNOWLEDGEMENT

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

REFERENCES

[1] L. V. Todd, "C++ templates are turing complete," *Available at citeseer.ist.psu.edu/581150.html*, 2003.
[2] D. R. Gabriel, B. Stroustrup, and J. Maurer, "Generalized constant expressions–revision 5," ISO SC22 WG21 TR, Tech. Rep., 2007.
[3] (2023) ClangOz. [Online]. Available: https://github.com/agozillon/ClangOz
[4] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008.
[5] H. Alblas, R. op den Akker, P. O. Luttighuis, and K. Sikkel, "A bibliography on parallel parsing," *ACM Sigplan Notices*, vol. 29, no. 1, pp. 54–65, 1994.
[6] 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.
[7] V. Seshadri, S. Weber, D. Wortman, C. Yu, and 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.
[8] 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.
[9] T. Gross, A. Sobel, and 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.
[10] I. J. S. 22, *ISO/IEC 14882:2020 Programming languages — C++*, 2020.
[11] P. Dimov, L. Dionne, N. Ranns, R. Smith, and D. Vandevoorde, "More constexpr containers," 2019. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0784r7.html
[12] A. Sutton, "C++ extensions for concepts," 2017. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0734r0.pdf
[13] H. Sutter, "Metaclasses: Generative c++," 2018. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0707r3.pdf
[14] M. Chochlík, A. Naumann, and D. Sankel, "Static reflection," 2017. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0707r3.pdf
[15] H. Dusíková. (2016) Compile time regular expressions. [Online]. Available: https://github.com/hanickadot/compile-time-regular-expressions
[16] M. Sánchez. (2018) tinyrefl. [Online]. Available: https://github.com/Manu343726/tinyrefl
[17] N. J. Bouman, "Multiprecision arithmetic for cryptology in c++ - compile-time computations and beating the performance of hand-optimized assembly at run-time," arXiv:1804.07236, 2018, https://arxiv.org/abs/1804.07236.
[18] B. Fahller. (2017) lift. [Online]. Available: https://github.com/rollbear/lift
[19] G. Steele, *Common LISP: the language*. Elsevier, 1990.
[20] A. Alexandrescu, *The D programming language*. Addison-Wesley Professional, 2010.
[21] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.
[22] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.
[23] C. McCord, *Metaprogramming Elixir*, 1st ed. Pragmatic Bookshelf, 2015.
[24] S. Baxter. (2020) Circle: The c++ automation language. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/
[25] N. D. Jones, "An introduction to partial evaluation," *ACM Computing Surveys (CSUR)*, vol. 28, no. 3, pp. 480–503, 1996.
[26] T. L. Veldhuizen, "C++ templates as partial evaluation," *arXiv preprint cs/9810010*, 1998.
[27] A. Tyurin, D. Berezun, and 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.
[28] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt, "Anydsl: A partial evaluation framework for programming high-performance libraries," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.
[29] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and 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.
[30] C. Consel and O. Danvy, "Partial evaluation in parallel," *Lisp and Symbolic Computation*, vol. 5, no. 4, pp. 327–342, 1993.
[31] M. Sperber, P. Thiemann, and H. Klaeren, "Distributed partial evaluation," in *Proceedings of the second international symposium on Parallel symbolic computation*, 1997, pp. 80–87.
[32] A. Bouter, T. Alderliesten, A. Bel, C. Witteveen, and 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.
[33] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.
[34] I. J. S. 22, "Technical specification for c++ extensions for parallelism," Tech. Rep., 2018.
[35] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.
[36] C. Bienia, S. Kumar, J. P. Singh, and 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.
[37] D. Bagley. (2001, Apr.) The computer language benchmarks game. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/