Metaprogramming-Enabled Parallel Execution of Apparently Sequential C++ Code

David S. Hollman*, Janine C. Bennett[†], Hemanth Kolla[‡],
Jonathan Lifflander[§], Nicole Slattengren[¶] Jeremiah Wilke[‖]
Sandia National Labs
Scalable Modeling and Analyis
Livermore, CA, 94550
Email: *dshollm@sandia.gov, [†]jcbenne@sandia.gov, [‡]hnkolla@sandia.gov,
[§]jjliffl@sandia.gov, [¶]nlslatt@sandia.gov, [‖]jjwilke@sandia.gov

***Abstract*—
Task-based execution models have received considerable attention in recent years to meet the performance challenges facing high-performance computing (HPC). In this paper we introduce MetaPASS — *Meta*programming-enabled *P*ara-llelism from *A*pparently *S*equential *S*emantics — a proof-of-concept, non-intrusive header library that enables implicit task-based parallelism in a sequential C++ code. MetaPASS is a data-driven model, relying on dependency analysis of variable read-/write accesses to derive a directed acyclic graph (DAG) of the computation to be performed. MetaPASS enables embedding of runtime dependency analysis directly in C++ applications using only template metaprogramming. Rather than requiring verbose task-based code or source-to-source compilers, a native C++ code can be made task-based with minimal modifications. We present an overview of the programming model enabled by MetaPASS and the C++ runtime API required to support it. Details are provided regarding how standard template metaprogramming is used to capture task dependencies. We finally discuss how the programming model can be deployed in both an MPI+X and in a standalone distributed memory context.**

***Index Terms*—task-based runtimes, template metaprogramming, parallel computing, programming models, execution models**

I. Introduction

Task-based execution models have received considerable attention in recent years as a mechanism for improving the performance of science and engineering codes. Examples include node-level tasking systems for use with MPI [21] applications (i.e., the X in MPI+X) [6] as well as holistic distributed memory task-models that completely replace MPI (e.g., Legion [7], HPX [25]). In both scenarios, a computational directed acyclic graph (CDAG) [20] captures dependencies between tasks.[1] The CDAG provides the runtime system with the ability to execute asynchronously and to perform application *lookahead*, often referred to as deferred execution. Lookahead enables performance-improving transformations of a default sequential schedule (i.e., in-order reading of the source code):

- Prefetching: Memory/remote accesses can be executed in advance, hiding communication latency to avoid stalls in the computational pipeline.
- Out-of-order execution: Tasks can run immediately when inputs are available to avoid stalls in the compute pipeline or be re-ordered to maximize data reuse.
- Move tasks to data: Tasks can be flexibly scheduled to the compute resource closest to the required data.

[1]We define a task here only as a group of related operations or instructions that, once scheduled, must be executed together on a single compute resource.

These transformations mitigate critical challenges posed by extreme-scale architectures by enabling optimizations such as exposing maximal parallelism to effectively leverage ever-increasing on-node compute resources, managing locality and staging of data across deep memory hierarchies, and hiding communication latency.

The creation and scheduling of a task-based CDAG for a particular application requires the following properties to be defined or determined at some level of the software stack:

1) Task granularity: How many statements should be grouped together within a task?
2) Task dependencies: What ordering dependencies between tasks must be preserved?
3) Task placement and scheduling: Where and when should a task execute?

In most existing tasking systems, task granularity (Item 1) is chosen *a priori* by the application developer. There is a vast amount of research on task placement and scheduling (Item 3), with proposed solutions that span the spectrum of placing the burden of responsibility on application developer, compiler, and runtime system. The current work focuses on Item 2: dependency capture and analysis.

Broadly, task-based programming models are either execution-driven, requiring explicit fork/join statements to create asynchronous work, or data-driven, implicitly creating asynchronous work when data usages do not conflict. Data-driven systems require dependency analysis to implicitly derive which tasks may run in parallel. This occurs either at compile-time for auto-parallelizing compilers [32] or at runtime with libraries like Legion [7]. Data-driven models often rely on sequential semantics to derive a concurrency specification. When data usages conflict, the task appearing first in program order must execute first. In execution-driven models, on the other hand, "dependency analysis" is essentially performed by the application developer since he or she chooses which tasks are parallel.

The precursor to dependency analysis for data-driven models is *dependency capture* - associating dependencies and their access modes with a given task. Even if the runtime system automatically performs dependency analysis, it either requires compiler modifications to auto-generate capture hooks, a new language (for instance, in the case of Regent [34]), or application developers to explicitly enumerate dependencies. Enumerating dependencies requires task pragmas in OpenMP [30] or creating region requirements in Legion [7], which leads to verbose and arguably difficult to maintain code (sometimes with loss of type-safety) that differs significantly from a sequential C/C++ code.

Here we introduce a proof-of-concept header library named MetaPASS (*Meta*programming-enabled *P*arallelism from *A*pparently *S*equential *S*emantics) that can embed dependency analysis and deferred execution directly in sequential C++ codes by transparently creating dependency capture hooks without requiring a complicated source-to-source compiler or compiler extension. Template metaprogramming alone is leveraged, producing user-level C++ code that is almost indistinguishable from a sequential application. As a proof-of-concept, MetaPASS has an extremely simple user interface. It introduces only two user-level constructs: a class template `async_ptr` and an `async` function. The MetaPASS header library is non-intrusive and modularly designed, allowing existing C++ codes to be adapted and integrated with existing runtime dependency analysis libraries and task-based runtime systems.

We begin in Section II by introducing MetaPASS along with existing C++ concurrency features, comparing to related work in Section III. Section IV shows example applications written using MetaPASS. Section V explains the software stack and runtime required to support the programming model. Section VI explains the C++ parameter detection idiom used to capture task dependencies without extending the compiler. Finally, Section VII discusses how the programming model can be used in an MPI+X or standalone distributed memory context, particularly addressing whether sequential semantics itself can be scalable.

II. MetaPASS and C++ Concurrency Class Templates

Asynchronous tasks, data-driven futures, and the thread-safe `std::shared_ptr` templates (for multi-threaded garbage collection) are now part of the C++ standard library [2]. C++ concurrency features leverage lightweight class templates wrappers around existing types that "append" functionality - reference counting in the case of `std::shared_ptr` and asynchronous task creation with `std::future`. A simple use of `std::future` and `std::async` is shown in Figure 1, which allows integers to be computed asynchronously and in parallel. While these C++ features enable several powerful programming patterns, they require a blocking `get()` call to access values. Additionally, they carry no information on whether the future value will be used for reading or writing, and thus support execution-driven task models only.

```
int taskA();
int taskB();
void print(int a, int b);
std::future<int> a = std::async(taskA);
std::future<int> b = std::async(taskB);
print(a.get(), b.get());
```

Fig. 1: Basic use of C++ `std::future` and `std::async`

Following the C++ standard philiosophy, MetaPASS provides `async_ptr`, a type-safe, lightweight template wrapper that enables a data-driven task model. Figure 2 shows the MetaPASS code corresponding to the simple example from Figure 1. We highlight that no blocking `get()` calls are required and `async_ptr` need not even appear in the function prototypes. When tasks run, MetaPASS can transparently extract the value. To preserve sequential semantics, the `print` function should execute after `taskA` and `taskB`. `taskA` and `taskB` operate on different variables and hence may safely run in parallel. *Dependency analysis* first requires *dependency capture* of the variables used to derive ordering constraints (i.e. the CDAG). OpenMP and Legion require dependencies to be explicitly enumerated and tagged as read or write. The C++ type system already distinguishes pass-by-value, pass-by-reference, `const`, and non-`const` parameters. C++ type traits can therefore distinguish read-only copies, read-only in-place, and read-write uses. `taskA` requires a non-`const` reference indicating read-write access. `print` takes two arguments by value, indicating a read-only copy.

```
using namespace mpass;
void taskA(int& a);
void taskB(int& b);
void print(int a, int b);
/* ... */
async_ptr<int> a, b;
async(taskA, a);
async(taskB, b);
async(print, a, b);
```

Fig. 2: A simple example using MetaPASS, which enables a data-driven task model.

```
spawn TaskA(a);
spawn TaskB(b);
sync;
spawn print(a,b);
```

Fig. 3: An execution-driven transformation would require that all decisions regarding when it is safe to `spawn` and `sync` work must occur at compile-time.

The MetaPASS header library provides metaprogramming-based transformations of application code, allowing dependency capture to occur transparently. These transformations occur at compile-time through template metafunctions that perform functionality similar to a source-to-source compiler (but are fully embedded in standard C++). To illustrate, the code in Figure 2 could be transformed in an execution-driven manner (Figure 3) or a data-driven manner (Figure 4). The transformation to the execution-driven code in Figure 3 requires the compiler to perform dependency analysis, deciding at compile-time when `spawn` is safe and when `sync` must occur to preserve sequential semantics. Such transformations have been explored in the context of auto-parallelizing compilers [32]. The data-driven transformation does not create explicit `spawn`/`sync` statements, rather the transformation inserts *dependency capture* hooks, but still requires runtime *dependency analysis*. MetaPASS provides transformation functionality equivalent to that in Figure 4 by directly leveraging the C++ language without requiring any compiler modifications or extensions.

```
Task A(taskA, 1); //task with 1 dep
A.add_dependency(a, ReadWrite);
Task B(taskB, 1); //task with 1 dep
B.add_dependency(b, ReadWrite);
Task print(print, 2); //task with 2 deps
print.add_dependency(a, Read);
print.add_dependency(b, Read);
```

Fig. 4: A data-driven transformation with *dependency capture* hooks instead of explict `spawn`/`sync` statements, rather the transformation inserts *Dependency analysis* is still required at run-time.

In summary, MetaPASS enables the following:

- Implicit parallelism from sequential semantics with a non-blocking model
- Type-safety for all task arguments
- Dependency analysis across translation units (since analysis occurs at runtime)
- Read/write access modes automatically inferred from type traits without explicit annotations

III. Related Work

Legion is a data-driven framework that implements runtime dependency analysis based on sequential semantics and read/write access requests on so-called logical regions [7]. Logical regions are similar to `async_ptr` class templates since they wrap an underlying data structure and track usages. However, logical region substructure is specified in terms of Legion-specific index and field spaces instead of C++ types. Regent is a Lua-based higher-level language that generates code for the underlying Legion C++-runtime, which reduces much of the verbosity of the Legion C++ interface [34]. In both C++ and Regent, access modes of logical regions must be explicitly marked in each task. OpenMP [30], [5] and OmpSs [11] are pragma-based tasking frameworks that provide a `#pragma task` for which in, out, an inout dependencies can be declared.

HPX [25], [22] provides futures and other local control objects for controlling program execution. While HPX implements type-safe task objects and is amenable to dataflow algorithms, it does not support sequential semantics with automatic dependency analysis, instead relying on explicit synchronization constructs. PARSEC (previously DaGuE) similarly supports data-flow applications [35]. While individual kernels in PARSEC can be implemented in C/C++/Fortran, the code generating the DAG is usually given in a PARSEC-specific JDF (job description format). Other notable examples of data-driven frameworks include TASCEL/Scioto [18], [26] and Deterministic Parallel Java [31].

Many execution-driven tasking frameworks exist that provide explicit async-finish or fork-join keywords for creating tasks, inluding Cilk [9], X10 [15], and Habanero Java [13]. Another example of a runtime system requiring explicit CDAG creation is OCR [27], where task dependencies are explicitly defined through event structs. In all of these systems task creation and synchronization is done explicitly.

Asynchronous tasks, data-driven futures, and the thread-safe `shared_ptr` templates (which manage multi-threaded garbage collection) have been part of the C++ standard [2] for some time now. One significant advantage of C++ concurrency features is that their associated C++ class templates are lightweight wrappers around user-defined types. This contrasts with systems like Legion (which uses index/field space constructs), MPI datatypes [12] (which uses vector/struct type creation calls and runtime-specific type descriptors), or OCR (which requires `void*` data blocks be cast to the correct type).

Domain-specific languages can also enable implicit parallelism. Uintah implements patch-based structured adaptive mesh refinement [33]. A sequential "patch-specific" code is implemented and the runtime system automatically derives task parallelism within a patch and implements data parallelism across patches. Listzt defines operations on vertices, edges, faces, and cells, which the Liszt framework then compiles into a task-based execution [17].

```
using namespace mpass;
using MatrixBlock = std::vector<double>;
void square_dgemm(int n,  MatrixBlock const& a,
   MatrixBlock const& b, MatrixBlock& c);
/* ... */
init();
async_ptr<MatrixBlock> A[N][N];
for (int i=0; i < N; ++i){
 async(dpotrf, n, A[i][i]);
 for (int j=i+1; j < N; ++j){
  async(dtrsm, n, A[i][i], A[j][i]);
 }
 for (int j=i+1; j < N; ++j){
  async(dsyrk, n, A[j][i], A[j][j]);
  for (int k=j+1; k < n; ++k){
    async(square_dgemm, n, A[j][i], A[k][i], A[j][k]);
  }
 }
}
finalize();
```

Fig. 5: A Cholesky decomposition example written using the MetaPASS programming model.

UPC++ [37] and Chapel [14] also provide tasking. In contrast to MPI+X, they do not explicitly separate distributed/on-node programming and often involve parallelism over implicitly distributed arrays. Implicit parallelism has been pursued through auto-parallel compilers [32]. We cannot cover the entire field and instead refer the reader to notable examples such as R-stream [28] and Pluto [10], which rely heavily on polyhedral analysis for auto-parallelization of loops [23]. Similar to the work here, template meta-programming has been used in Kokkos [19] and Raja [24] for performance-portable loop parallelization.

IV. MetaPASS Examples

A. Cholesky Decomposition

To illustrate the programming model enabled by the MetaPASS library, we consider the $N \times N$ tile-based, right-looking Cholesky decomposition [35] for blocks with $n \times n$ elements. The MetaPASS code is shown in Figure 5. Similar to C++ futures, asynchronous tasks are created by passing a function and its arguments to an `async` function. In contrast to an `std::async` and `std::future`, which create asynchronous tasks without any dependency analysis, MetaPASS will detect the access modes of all variables used and pass the information to a runtime dependency analysis layer. The non-blocking Cholesky code in Figure 5 is not possible with C++ futures, and would require tedious insertion of synchronization or `get()` calls to ensure a correct algorithm (to say nothing of a maximally performant one).

MetaPASS distinguishes task *arguments* (regular C++ arguments) from task *dependencies* (`async_ptr` arguments). In MetaPASS, task creation via `async` can mix arguments and dependencies. Note that the function prototype parameters for Cholesky do not need to distinguish between regular arguments and `async_ptr` arguments; however, at the call sites, the metaprogramming layer distinguishes between the integer argument `n` and the `async_ptr` dependencies. For `square_dgemm`, there are three dependencies and one argument at the call site. All arguments are copied since the corresponding parameters in the function prototype have pass-by-value semantics, whereas the dependencies to this function are pass-by-reference, indicating that the argument should be a asynchronously referenceable entity (in this case, an `async_ptr`). In this way, task parameters may be arguments at one `async` invocation and may be dependencies at another `async` invocation. This symmetry of arguments and dependen-

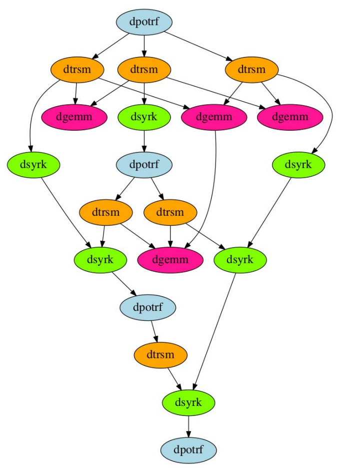

Fig. 6: CDAG illustrating task dependencies for the tile-based Cholesky decomposition example in Figure 5.

```
using namespace mpass;
void stencil(int nGrid,const Array& grad,
  Array& x_grd);
void chemistry(int nGrid,const Array& Y,
  const Array& T,const Array& p,Array& omega);
void updateRho(int nGrid,const Array& rho_grd,
  const Array& u_grd, Array& rho);
  ...
using ArrayWrapper = async_ptr<Array>;
ArrayWrapper rho, u, p, T, Y, omega;
ArrayWrapper rho_grd,u_grd,p_grd,T_grd,Y_grd;
init();
async(initial_values, rho, up, T, Y, omega);
for (int t=0; t < nStep; ++t){
  async(stencil,nGrid,rho,rho_grd);
  async(stencil,nGrid,u,u_grd);
  async(stencil,nGrid,p,p_grd);
  async(stencil,nGrid,T,T_grd);
  async(stencil,nGrid,Y,Y_grd);
  async(updateRho,nGrid,rho_grd,u_grd,rho);
  async(updateVel,nGrid,rho_grd,u_grd,p_grd,u);
  async(chemistry,nGrid,Y,T,p,omega);
  async(updateY,nGrid,rho_grd,u_grd,Y_grd,omega,Y);
  async(updateT,nPoints,rho_grd,u_grd,p_grd,T_grd,T);
  async(updateP,nPoints,rho,T,Y,p);
}
finalize();
```

Fig. 7: An iterative 1D-stencil chemistry example written using the MetaPASS programming model.

cies in the function prototype is (as far as we know) unique to MetaPASS.

Figure 6 shows the CDAG generated for a 4x4 tiled algorithm, showing task-ordering dependencies. The dense Cholesky shown here is not data-dependent aside from the structure parameters n and N. However, the dependency analysis still occurs at runtime. MetaPASS only captures variable *uses* and emits calls to a dependency analysis layer. Execution of tasks and dependency analysis/CDAG generation can be occurring concurrently.

B. Computational fluid dynamics

Computational fluid dynamics (CFD) with chemistry can introduce much more extensive CDAGs [16], [8]. They are also often iterative, introducing anti-dependencies with write permissions (non-const reference) waiting for a reader (const reference) to complete. A simple 1-D example is shown in Figure 7. We again see that function prototypes do not have to refer to `async_ptr` objects. The combination of const/non-const and value/reference determines whether a variable is a read-only copy, read-only in-place, or read-write dependency. In this example each function takes an initial argument (an integer number of points on the grid) followed by dependencies. Capturing variables and access modes with sequential semantics produces the CDAG in Figure 8, showing how data usages induce task ordering constraints (both dependencies and anti-dependencies).

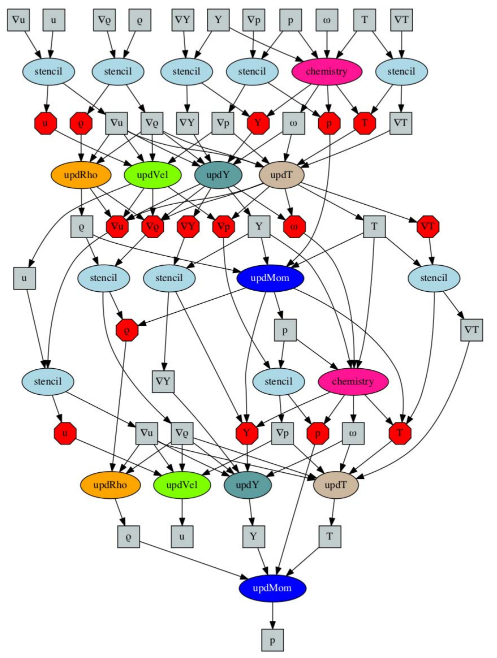

Fig. 8: Data-flow CDAG with dependency edges between data vertices (squares) and task vertices (circles) for the 1D-stencil chemistry example from Figure 7. Anti-dependency constraints are shown in red octagons.

V. MetaPASS Software Stack

The overall software stack for an application written using MetPASS is shown in Figure 9. Like Boost [1] or the C++ Standard Template Library, MetaPASS is a header-only library containing the `async_ptr` class template and corresponding template metaprogramming features. As discussed in Section II, the template layer creates hooks into a runtime dependency analysis layer. At run-time, the enforcement of sequential semantics occurs within the dependency analysis layer. Once dependency analysis is complete for a given task, it can be registered with a scheduler. The MetaPASS layer does not require any thread-safety aside from atomic integers for reference counting. In contrast, the dependency analysis layer must safely run in parallel. Once multiple tasks begin running on different threads, the adding or clearing of the same variable (dependency) from multiple tasks might occur simultaneously.

The runtime implementation could leverage several different libraries, for example using async/join calls in Cilk, Event Driven Tasks (EDTs) in OCR, or explicit pThreads or std::threads. Dependency analysis [36] and corresponding runtime implementations have been described in detail elsewhere. Here we emphasize the C++ concepts enabling type-safe deferred execution and delay a complete description of the dependency analysis and scheduler using MetaPASS for later work. The focus here is providing an "STL-inspired" C++ programming model for task-based execution that is compatible with existing runtime infrastructure. The remainder of this section provides additional details regarding the C++ runtime API required to support the MetaPASS programming model.

A. Header Library

As a proof-of-concept library, MetaPASS provides a very minimal interface. Calls to `async` create task objects. To avoid template code in the dependency analysis and runtime layers, tasks have a simple abstract interface:

```
struct TaskBase {
  virtual void run() = 0;
  std::vector<Dependency> dependencies;
};
```

Each `Dependency` is assigned a unique ID by the header library. The header library register tasks with the dependency analysis layer via the function:

```
void register_task(TaskBase&& task);
```

The function uses C++-11 move semantics, indicating "transfer of ownership" to the dependency analysis layer. The IDs and read/write access mode of all variables in the task is contained in the vector of dependencies, providing all the information needed for dependency analysis.

The concrete `Task` class template in MetaPASS uses variadic templates to store all function arguments in a `std::tuple`. This critical step enables deferred execution (and hence task-based lookahead). Rather than executing immediately, the function arguments are stashed in a `std::tuple` (making copies only as necessary, based on the call-site argument type and the function parameter type traits). When dependencies are satisfied, `task->run()` can be invoked, which unpacks arguments (potentially from dependencies) and passes them to the function.

B. Dependence Inference: Call Arguments and Function Parameters

The class declaration above does not show how dependenices are actually inferred. C and C++ allow implicit type conversions. The variable type passed to a function may not exactly match the function prototype. Both the call-site type and the function parameter type are required for dependency capture. Table I details the semantics of C++ variable captures. Table I shows that MetaPASS is actually compatible with multi-level tasking, the details of which we delay to later work. Functions may take `async_ptr` parameters, which indicates that they may themselves schedule more tasks dependent on that parameter.

VI. Parameter Detection Idiom

To embed the above programming model in C++ without compiler modifications, we need several C++ template metafunctions that detect if the function parameter[2] is by value, a const reference, or a non-const reference. As described above, the first two of these indicate read-only usage, while the last indicates modify usage. If we wish to support multilevel tasking, we also need to be able to detect when

[2]The distinction between parameter and argument is important here. "Parameter" is a formal parameter in the function definition while "argument" is a variable given at the call site.

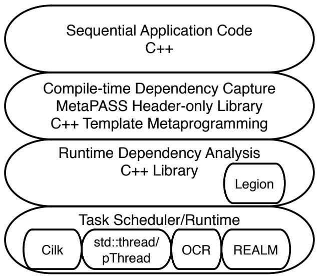
Sequential Application Code
C++
Compile-time Dependency Capture
MetaPASS Header-only Library
C++ Template Metaprogramming
Runtime Dependency Analysis
C++ Library
Legion
Task Scheduler/Runtime
Cilk
std::thread/
pThread
OCR
REALM

Fig. 9: Basic software stack showing transformation of sequential C++ code into task-based execution. Example libraries that might be integrated with MetaPASS are shown.

a function requests permission to schedule tasks dependent on an argument — that is, when it has a parameter of the form `async_ptr<T>`. For the first three of these requirements, we define the metafunctions `parameter_N_is_by_value`, `parameter_N_is_const_lvalue_ref`, and `parameter_N_is_nonconst_lvalue_ref`, all of which are described in Section VI-C2.

A. Parameter Detection and SFINAE

Much of modern C++ template metaprogramming is based on the Substitution Failure Is Not An Error (SFINAE) principle — that is, when the compiler creates a candidate set of function overloads or class template instantiations by substitution of template arguments, candidates that would lead to errors are ignored.[4, §14.8.2] For instance:

```
template <class T> void func(T) { }
template <class U> void func(typename U::type) { }
struct A { typedef double type; };
int main() {
  func<A>(3.14); // calls 1st overload
  func<int>(25); //calls 2nd overload
  //failed substitution on 2nd overload isn't an error
  //even though 'typename int::type' is ill-formed
}
```

We can detect if a valid function overload `func` exists:

```
// Some functions to do detection on:
double func(int);
int func(std::string const&);

// helper, needed for reasons explained below
template <class> struct wrap { typedef void type; };

// Metafunction that "detects" if func can be
// called with an object of type T
template <class T, class Enable=void>
struct func_valid_for { enum { value = false }; };

// This specialization takes advantage of SFINAE:
template <class T>
struct func_valid_for<T, test substitution func(T)>
{ enum { value = true }; };
/* ... */
func_valid_for<int>::value; // true
func_valid_for<long>::value; // true
func_valid_for<std::string>::value; // true
func_valid_for<std::vector<int>>::value; // false
```

with pseudocode inserted. Because of rules for partial ordering of class template specializations,[4, §14.5.5.2] any type `T` that does *not* lead to a substitution failure for the second version of `func_valid_for` will prefer that specialization, and have an enumeration member `value` equal to `true`. That will only happen if it is valid call `func` with objects of type T. The test substitution pseudocode can be written out:

```
typename wrap<decltype(func(std::declval<T>()))>::type
```

which will be a valid substitution if the function overload exists and a failure otherwise (the details of `decltype` and

Call Argument Type	Function Parameter Type				
	By-value	Reference	`const` Reference	`async_ptr<T>`	`async_ptr<T const>`
`async_ptr<T>`	Read-write copy dependency	Read-write in-place dependency	Read-only in-place dependency	Read-write schedule dependency	Read-only schedule dependency
`async_ptr<T const>`	Read-only copy dependency	Compile-time error	Read-only in-place dependency	Compile-time error	Read-only schedule dependency
lvalue	Read-write copy argument	Compile-time error	Compile-time error	Compile-time error	Compile-time error
rvalue	Read-write copy argument	Read-write copy argument	Read-only copy argument	Compile-time error	Compile-time error

TABLE I: How read-only/read-write and argument/dependency properties are inferred from variables types at the call site and parameter types in the function.

`declval` are not needed here and the interested reader can consult the C++ reference). SFINAE is the basis for a number of widely used C++ idioms, most notably the `enable_if` idiom (incorporated into namespace `std` in C++11 onwards), which allows the selection of function overloads or class template partial specializations based on arbitrary properties of template arguments.

The simple form of `func_valid_for` doesn't work for arbitrary functions, functors, or other callables[3] with arbitrary number of arguments. Making this generalization requires the use of variadic template parameters, which is a feature available in C++11 onwards. We can also make the generalization to an arbitrary callable by making it a template parameter as well. For instance:

```
template <class Callable, class Enable, class... Args>
struct detect_valid : std::false_type { };

using std::declval;
template <class Callable, class... Args>
struct detect_valid<
  Callable,
  typename wrap<
    decltype(declval<Callable>()(declval<Args>()...))
  >::type,
  Args...
> : std::true_type { };
```

Inheriting from `std::false_type` and `std::true_type` is the preferred way of setting `value` to `false` or `true`. We can then use `detect_valid` the same way we used `func_valid_for`:

```
void f1(int);
std::string f2(float);
struct Functor { double operator()(double, int); };
/* ... */
detect_valid<decltype(f1), void, int>::value // true
detect_valid<decltype(f2), void, float>::value // true
detect_valid<Functor, void, double, int>::value // true
```

This can be cleaned up using a simple alias:

```
template <class F, class... Args>
using is_valid = detect_valid<F, void, Args...>;
```

A very simple generalization on this theme leads to the `void_t`-based detection idiom, first popularized by Walter E. Brown and accepted as an extension to the C++ standard library for the C++17 standard.[3]

B. Reverse Template Argument Deduction

The idiom discussed above provides much of the framework needed for detecting the validity of specific arguments when their type and number are known. For our programming model, we need something more general. Most C++ developers are familiar with template argument deduction:

```
template <class T>
void my_func(T const& arg) { }
/* ... */
my_func(42); // T deduced as int, arg is int const&
std::string my_str = "hello";
my_func(my_str); // T deduced as std::string, arg
                 // is std::string const&
```

The compiler pattern matches the type of the call argument (`int` in the first, `std::string&` in the second) with the type expression of the function parameter (`T const&`) to determine the type `T` for substitution.[4] This process can be made to happen backwards:

```
struct backwards {
  template <class T>
  operator T const&();
};
void f1(std::string const&);
void f2(double);
```

When the compiler encounters a call like `f1(backwards())`, the C++ standard states[4, §14.8.2.3] that template argument deduction works basically the same way, but *backwards*: the compiler pattern matches the function parameter (`std::string const&` for `f1`) with the type expression in the conversion operator (`T const&` in this case).

C. Generalized Parameter Type Detection

The basic strategy for the parameter detection metaprogramming toolkit is as follows: 1) provide a metafunction that can count the number of arguments (using a class with a permissive templated conversion operator), then 2) replace the Nth argument with a class with a restrictive templated conversion operator, and finally 3) use the call detection idiom to determine the validity of making a call of the callable in question with the replacement argument.

1) Counting Parameters: To count the parameters to a callable, we first need an argument that we can count on to be a valid argument for *any* parameter. For our purposes, there are four main catagories of parameters we need to worry about: by-value parameters (`T` or `T const`), non-`const` lvalue references (`T&`), `const` lvalue references (`T const&`), and rvalue references (`T&&`). The class that we need for this purpose is

```
struct any {
  template <class T> operator T();
  template <class T> operator T&() const;
};
```

Consider each of the categories relevant here. For non-`const` lvalue reference parameters (for instance, `int&`), the first overload clearly fails deduction since a temporary can't bind to a non-`const` reference. The second overload works. Similarly, for rvalue reference parameters like `int&&`, the second overload fails because a lvalue can't bind to an rvalue reference, but the first overload (which generates an rvalue) works. The `const` lvalue reference case is a bit more complicated, since either template could generate a type that can bind to a `const`

[3]A "callable" here refers to functions, lambdas, and classes (termed functors) with a call operator `operator()`

[4]The exact rules for template argument deduction are quite extensive,[4, §14.8.2][29]).

lvalue reference, but the compiler prefers `const` conversion of a reference to the generation of a temporary,[4, §8.1.3/(5.1.2)] thus imposing a partial order on the generated candidates. Now consider a by-value parameter like `int`. The first overload works,[5] but the second overload *also* works. This ambiguity would normally lead to a substitution failure, except we've marked the second conversion operator as a `const` member function of `any`, thus imposing a partial ordering on these two candidates (i.e., if the instance of `any` on the call side is `const`, the second will be preferred; otherwise, the first will be preferred).

With the form of the "universal argument" `any` established, it is relatively simple to write a recursive metafunction that counts arguments of a given callable (using the `is_valid` metafunction from above):[6]

```
template <class F, class... Args>
struct count_parameters
  : std::conditional<
      is_valid<F, Args...>::value,
      std::integral_constant<size_t, sizeof...(Args)>,
      count_parameters<F, Args..., any>
    >::type::type
{ };
```

where `std::conditional` is a metafunction from the C++ standard library that selects the second or third argument depending on the value of the first (boolean) template argument. The double `::type::type` allows the recursion to short-circuit properly, so that arbitrarily many `count_parameters` instantiations aren't generated.

2) Replacing the Nth Argument: To actually garner useful information about the callable's parameters, we need both (A) more restrictive analogs of `any` that work for some (but not all) of our parameter "catagories" of interest (or that test for some other type property of interest), and (B) a way to probe each parameter individually by substituting these more constrained analogs into the Nth argument slot and testing for validity of the call. The latter of these two can be done using a recursive metafunction similar to `count_parameters`:

```
template<class F, size_t N, size_t I, size_t Tot,
  class sub_any, class... Args>
struct sub_N_helper
  : sub_N_helper<F, N, I+1, Tot, sub_any, Args...,
      typename std::conditional<
        I == N, sub_any, any
      >::type
    >::type { };
// recursive base case specialization:
template<class F, size_t N, size_t Tot,
  class sub_any, class... Args>
struct sub_N_helper<F, N, Tot, Tot, sub_any, Args...>
  : is_valid<F, Args...>  { };
// type alias to call the helper:
template <class F, size_t N, class constrained_any>
using is_valid_substitute_N = sub_N_helper<F, N, 0,
  count_parameters<F>::value, constrained_any>;
```

All that remains to develop a metafunction like `parameter_N_is_by_value<F, N>` is to come up with useful types to give for the `constrained_any` template argument. Eventually, we want to detect the four parameter categories (e.g., `int`, `int&`, `int const&`, and `int&&`) so that we can determine whether an argument's usage is read-only or read-write. We can start with the by-value case. As mentioned in Section VI-C1, the implementation of `any` in that section would not work (i.e., would generate two ambiguous candidates) if latter template function were not marked as `const`. We can use that to our advantage here:

```
struct ambiguous_if_by_value {
  template <class T> operator T();
  template <class T> operator T&();
};
// The Nth parameter is by value only if it *cannot* be
// called with ambiguous_if_by_value in the Nth slot:
template <class F, size_t N>
using parameter_N_is_by_value = std::integral_constant<
  bool, !is_valid_substitute_N<
    F, N, ambiguous_if_by_value
  >::value
>;
```

Next, we can handle `const` lvalue references:

```
struct any_const_lvalue_ref {
  template <class T> operator T const&();
};
template <class F, size_t N>
using parameter_N_is_const_lvalue_ref =
  std::integral_constant<bool,
    is_valid_substitute_N<F, N,
      any_const_lvalue_ref>::value
    && !parameter_N_is_by_value<F, N>::value
  >;
```

where we've explicitly excluded by-value parameters because they can also accept an argument of the form `T const&`. `parameter_N_is_rvalue_ref` is omitted here for brevity. Once these three are implemented, the implementation of `parameter_N_is_nonconst_lvalue_ref` is just the negation of the other cases:

```
template <class F, size_t N>
using parameter_N_is_nonconst_lvalue_ref =
  std::integral_constant<bool,
    !parameter_N_is_by_value<F, N>::value
    && !parameter_N_is_const_lvalue_ref<F, N>::value
    && !parameter_N_is_rvalue_ref<F, N>::value
  >;
```

VII. Distributed Memory

Using MetaPASS for MPI+X parallelism is straightforward. While future extensions may incorporate MPI calls into the dependency capture and analysis, the current version follows the OpenMP model of parallel regions. OpenMP uses C/C++ scope to begin/end regions. The current work uses `init` and `finalize` calls to begin and end a MetaPASS region. An alternative syntax could use C++11 lambdas to explicitly scope parallel regions without `init` and `finalize` calls, but we delay that for later work.

Rather than MPI+X mode, MetaPASS is compatible with HPX or Legion tasking that begins with a single-level top level task instead of many SPMD (single-program, multiple data) tasks as done in MPI. Sequential semantics would therefore cover both on-node and off-node parallelism. The critical challenge in such a unified programming model is the scalability of sequential semantics itself. Consider an SPMD-style loop launching a huge number of tasks corresponding to grid patches in a mesh (*i.e.*, N is large):

```
int top_level_task(int argc, char** argv){
  for (int patch=0; patch < N; ++patch){
    async(timestep, mesh.patches[N]);
  }
```

For this to be performant two issues must be addressed:

1) Dependency analysis must be distributed since a single process would quickly become a bottleneck if all analysis were performed in one place.
2) All mesh data cannot be allocated in the root task since the root process would quickly run out of memory.

MetaPASS helps in addressing the second issue. `async_ptr` works much like a `shared_ptr` and can therefore hold a null

[5] assuming the copy and move constructors of the parameter type aren't both deleted; if they are, the callable itself can't be invoked in any usual context anyway

[6] A more general version of `count_parameters` can quite easily be written to handle things like multiple overloads and default arguments, but is omitted here for brevity.

or stub value. Tasks can be scheduled to an `async_ptr` without allocating memory to large underlying arrays. After `timestep` tasks have been distributed, arrays for holding mesh data can be allocated in a scalable manner. For scheduling remote tasks or work-stealing, MetaPASS tasks critically know the exact type of all parameters.

The first issue (scalable dependency analysis) is more challenging and has received considerable attention in Legion [7]. Legion provides an index space launch over data partitions with special optimizations for disjoint partitions (hence no read/write conflicts between tasks). The index space launch is a programming model construct, leaving the mechanism unspecified and is hence compatible with a, *e.g.*, tree-based implementation. Rather than relying directly on C++ loops, MetaPASS would require another feature to indicate that the backend must support a scalable launch, *e.g.*

```
async_for(timestep, 0, N, mesh.patches);
```

VIII. Conclusions

In this paper we introduced the MetaPASS programming model, highlighting the ease with which it integrates into existing C++ application codes through examples. Our "STL-inspired" C++ programming model for task-based execution has a modular design and is compatible with existing runtime infrastructure (the header library will be made publicly available at http://darma.sandia.gov pending Sandia's software release process). Herein we have focused on the C++ concepts that enable type-safe deferred execution of tasks, describing in detail the template metaprogramming infrastructure comprising our header library. Future work will explore MetaPASS integration with existing dependency analysis and runtime system technologies, both in an MPI+X and a standalone context.

IX. Acknowledgments

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This work was supported by the U.S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) program.

References

[1] Boost C++ Libraries. http://www.boost.org.
[2] ISO/IEC 14882:2014(E) - Programming Language C++.
[3] ISO/IEC JTC1 SC22 WG21 N4436 - Proposing Standard Library Support for the C++ Detection Idiom. Technical Report N4436, Geneva, Switzerland, April 2015.
[4] ISO/IEC 14882:2014(E) - Programming Language C++ [Working Draft]. Technical Report N4594, Geneva, Switzerland, May 2016.
[5] E. Ayguadè et al. A Proposal for Task Parallelism in OpenMP. In *A Practical Programming Model for the Multi-Core Era*. Springer Berlin Heidelberg, 2008.
[6] R. F. Barrett et al. Toward an evolutionary task parallel integrated MPI + X programming model. In *PMAM '15: Programming Models and Applications for Multicores and Manycores*, pages 30–39, 2015.
[7] M. Bauer et al. Legion: expressing locality and independence with logical regions. In *SC '12: High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.
[8] M. Bauer et al. Structure slicing: Extending logical regions with fields. In *SC '14: High Performance Computing, Networking, Storage and Analysis*, pages 845–856, Piscataway, NJ, USA, 2014. IEEE Press.
[9] R. D. Blumofe et al. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Notices*, 30:207–216, 1995.
[10] U. Bondhugula et al. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI 2008: Programming Language Design and Implementation*, pages 101–113, 2008.
[11] J. Bueno et al. Productive Programming of GPU Clusters with OmpSs. In *IPDPS: 26th International Parallel & Distributed Processing Symposium*, pages 557–568, 2012.
[12] S. Byna et al. Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost. In *Cluster 2013*, pages 412–419, 2003.
[13] V. Cavè et al. Habanero-Java: the new adventures of old X10. In *PPPJ 2011: Principles and Practice of Programming in Java*, pages 51–61, 2011.
[14] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
[15] P. Charles et al. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA 2005: 20th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 519–538, 2005.
[16] J. H. Chen et al. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science and Discovery*, 2(1):015001, 2009.
[17] Z. DeVito et al. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *SC '11: High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.
[18] J. Dinan et al. Scioto: A Framework for Global-View Task Parallelism. In *ICPP 2008: 37th International Conference on Parallel Processing*, pages 586–593, 2008.
[19] H. C. Edwards and C. R. Trott. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *XSW 2013: Extreme Scaling Workshop* , pages 18–24, 2013.
[20] N. Fauzia et al. Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential. *ACM Trans. Archit. Code Optim.*, 10:1–29, 2013.
[21] M. P. I. Forum. *MPI: A Message-Passing Interface Standard: Version 2.1*. 2008.
[22] G. R. Gao et al. ParalleX: A Study of A New Parallel Computation Model. In *IPDPS 2007: 21st International Parallel and Distributed Processing Symposium*, pages 1–6, 2007.
[23] M. Griebl. *Loop Programs for Distributed Memory Architectures*. PhD thesis, Universität Passau, 2004.
[24] R. Hornung and J. Keasler. The RAJA Portability Layer: Overview and Status Tech Report LLNL-TR-661403. , 2014.
[25] H. Kaiser et al. HPX: A Task Based Programming Model in a Global Address Space. In *International Conference on Partitioned Global Address Space Programming Models*, pages 1–11, 2014.
[26] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Work Stealing and Persistence-Based Load Balancers for Iterative Overdecomposed Applications. In *HPDC '12: High-Performance Parallel and Distributed Computing*, pages 137–148, 2012.
[27] T. Mattson and R. Cledat. OCR: The Open Community Runtime Interface v1.1.0.
[28] B. Meister et al. R-Stream Compiler. In *Encyclopedia of Parallel Computing*. Springer US, Boston, MA, 2011.
[29] S. Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, 2014.
[30] S. L. Olivier et al. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *Int. J. High Perform. Comput. Appl.*, 26:110–124, 2012.
[31] J. Robert et al. A type and effect system for deterministic parallel Java. *SIGPLAN Not.*, 44:97–116, 2009.
[32] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPoPP '99: Principles and practice of parallel programming*, pages 72–83, 1999.
[33] J. Schmidt et al. Large Scale Parallel Solution of Incompressible Flow Problems Using Uintah and Hypre. In *CCGrid '13: Cluster, Cloud and Grid Computing*, pages 458–465, 2013.
[34] E. Slaughter et al. Regent: a high-productivity programming language for HPC with logical regions. In *SC '15: High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.
[35] F. Song et al. Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-Memory Multicore Systems. In *SC '09: High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2009.
[36] S. Treichler, M. Bauer, and A. Aiken. Language Support for Dynamic, Hierarchical Data Partitioning. In *OOPSLA 2013: Object Oriented Programming Systems Languages and Applications*, pages 495–514, 2013.
[37] Y. Zheng et al. Upc++: A pgas extension for c++. In *IPDPS 2014: International Parallel and Distributed Processing Symposium*, pages 1105–1114, May 2014.