Pygion: Flexible, Scalable Task-Based Parallelism with Python

Elliott Slaughter
SLAC National Accelerator Laboratory
eslaught@slac.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

***Abstract*—Dynamic languages provide the flexibility needed to implement expressive support for task-based parallel programming constructs. We present Pygion, a Python interface for the Legion task-based programming system, and show that it can provide features comparable to Regent, a statically typed programming language with dedicated support for the Legion programming model. Furthermore, we show that the dynamic nature of Python permits the implementation of several key optimizations (index launches, futures, mapping) currently implemented in the Regent compiler. Together these features enable Pygion code that is comparable in expressiveness but more flexible than Regent, and substantially more concise, less error prone, and easier to use than C++ Legion code. Pygion is designed to interoperate with Regent and can use Regent to generate high-performance CPU and GPU kernel implementations. We show that, in combination with high-performance kernels written in Regent, Pygion is able to achieve efficient, scalable execution on up to 512 nodes of the heterogeneous supercomputer Piz Daint.**

***Index Terms*—task-based parallelism, Pygion, Legion, Python**

I. Introduction

A growing class of users in the physical sciences and data analytics are unfamiliar with traditional high-performance programming models and languages, yet need access to high-performance computational resources. This issue is of particular relevance to science user facilities such as the Linac Coherent Light Source (LCLS) [1], which regularly host users with no formal training in computer programming and who need to process and analyze increasingly large volumes of data produced by scientific experiments [2]. In many cases, analyses must be written, debugged and executed on the fly as the experiment is in progress. These users typically program in Python, but still need a path to achieve productive, high-performance programming on current and future supercomputers.

Task-based programming models provide a promising path forward. Such models, which include Legion [3], PaRSEC [4] and StarPU [5], simplify the programming of heterogeneous supercomputers by providing sequential semantics. *Tasks* (or functions marked for parallel execution) are enumerated in program order. The arguments to tasks, and the *privileges* requested on those arguments (*read*, *write*, etc.) are passed through a dynamic analysis to compute a dependence graph between tasks that guides the parallel and distributed execution of the program. Every execution, even when parallel and/or distributed, is guaranteed to be consistent with the original sequential ordering of tasks, ruling out by construction a large class of potential parallel and distributed programming bugs (data races, deadlocks, etc.) that can occur in programs written in traditional programming models.

Task-based programs are conceptually simpler than ones written in traditional programming models, but for two reasons they can sometimes be longer than one might expect, especially when written in C++. First, task-based programs reify the data flows in a program, as any data to be used by a task must be explicitly identified in the task's arguments, along with a corresponding privilege. This reification is an essential feature that drives the other advantages of task-based systems. Second, certain low-level features of task-based runtime systems, which are necessary for performance, are exposed in the runtime APIs for these systems making programming at this level more complicated than necessary. These issues are challenging to address in a C or C++ API as these languages lack type system features necessary to capture the relevant programming system invariants [6], which in turn results in a C/C++ interface which is verbose, error-prone, and low-level compared to the conceptual model of task-based systems (see Section III).

Programming languages with dedicated support for task-based programming constructs can address this mismatch. For example, Regent [7] is a high-level programming language that supports the Legion programming model. The Regent type system enforces many of the low-level invariants required for writing correct Legion programs, and the compiler provides a number of critical optimizations that improve performance and scalability [8]. However the cost of enforcing these invariants in the Regent type system is that the compiler must necessarily be conservative, making it difficult or impossible to express certain code patterns in an application.

In this paper, we explore an alternative approach based on the dynamic programming language Python. A key observation is that many program analysis problems that are difficult or intractable to solve at compile time can be solved in a straightforward manner using dynamic analysis. Type system invariants enforced at compile time by the Regent compiler can be checked at runtime in Python. Many (though not all) of the important optimizations provided by Regent can also be provided in Python via dynamic program analysis. We have implemented five of the seven optimizations in [7], and one has been rendered irrelevant by unrelated improvements in Legion, leaving only one optimization that could not be implemented (requiring manual user annotation of certain tasks). This approach leads to task-based programming constructs that

are more flexible (due to the dynamic nature of the checks), while maintaining similar levels of expressiveness and brevity in application codes.

There are tradeoffs involved in this approach, particularly resulting from the use of dynamic analysis to perform program optimization. Though certain problems can be addressed through careful API design, the abstractions are necessarily leaky to a certain extent. These leaks are not normally visible in idiomatic Python code, but can become visible, for example, if the user explicitly checks the types of certain values generated by the API (see Section IV).

Though Python code is certainly slower than that produced by Regent's highly-optimized LLVM [9] backend, this does not necessarily impact overall application performance. In particular, the *control code* in a task-based program (i.e., the part that launches tasks) can run asynchronously from the rest of the code, and thus only needs to achieve an average throughput (in tasks per second) that is higher than the rate at which the system can execute those tasks. If the tasks themselves are highly-optimized (perhaps because they are written in another programming language or call an external library), then the overall application performance can still be high.

We present Pygion, an implementation of Python support for Legion, and evaluate its performance against three well-tuned Regent applications on up to 512 nodes of the Piz Daint supercomputer [10]. By reusing the existing, high-performance Regent kernels, these implementations are able to achieve high-performance execution on Piz Daint's GPUs. For two out of the three applications we consider, we show that Pygion achieves performance parity (within 2% in absolute performance of Regent) at all node counts, while a third achieves within 16%.

This paper makes the following contributions:

- Section II presents a set of high-level Pygion APIs for Legion and discusses how they compare with Regent and C++ Legion.
- Section III describes a strategy for lowering the high-level Pygion APIs to the Legion runtime and the dynamic analysis needed to track the necessary programming model invariants.
- Section IV discusses key optimizations provided by the Regent compiler and their corresponding implementations in Pygion. This section also discusses an optimization that cannot currently be implemented in Pygion due to intractability of the necessary dynamic analysis, and potential strategies for implementing them in the future.
- Section V evaluates the performance and scalability of Pygion against reference implementations in Regent on up to 512 nodes of the Piz Daint supercomputer.

Section VI discusses related work, and Section VII concludes.

II. Pygion Programming Interface

In this section we describe the design of Pygion, a high-level, Python-based interface for Legion. For the purposes of this and the following sections, *Legion* refers to a runtime system implemented as a C++ API that provides all the services (launching tasks, moving data, profiling, etc.) of the Legion programming model. *Regent* is a high-level, statically typed language that compiles down to the Legion API. The Regent compiler statically checks that programs observe the critical invariants of Legion, and also simplifies programming by automating and optimizing some aspects of the programming model. Finally as noted above, *Pygion*, the subject of this paper, is a Python interface to the Legion API that, as we will see, provides most of the functionality of Regent running in a standard Python interpreter.

Writing a parallel program in Legion consists of two interrelated objectives: the user must divide the program execution into tasks (to be executed in parallel) and the program data into *regions* (to be distributed across the machine).

Tasks are simply functions, marked by the user as being eligible for parallel execution. The body of a task executes sequentially, but concurrently with other tasks. Tasks can expose nested parallelism to the Legion runtime by invoking *subtasks*. Subtasks run asynchronously, but always in a manner consistent with a sequential execution of the program.

To identify the parallelism in a Legion program, the runtime performs a dynamic analysis over the sequence of tasks to compute a dependence graph. A dependency exists between two tasks if they would *interfere*: i.e., when the two tasks access overlapping data, and at least one task has requested write privilege on the data. Note that privileges in Legion are *strict*. A task cannot access data unless it has been passed the data as an argument with the appropriate privilege.

Application data in Legion is stored in regions. Regions can be thought of as being similar to Pandas dataframes [11], but natively support multi-dimensional indexing. They contain *fields*, each of which stores a value for each point in an *index space*. In Pygion each field is exposed as a separate NumPy [12] array.

Listing 1 shows a simple SAXPY example program with two tasks. Tasks in Pygion are declared via the `@task` decorator (lines 1, 5). Privileges are specified via the `privileges` keyword as a list with one entry per argument. In this case, the task reads and writes the field `y` and reads the field `x` of the first argument `s` (line 1). The actual computation is performed on line 3 using NumPy array operations.

Execution starts at the task `main` (lines 5-10). `main` defines a region with 10 elements and two fields `x` and `y` (line 7), *partitions* it into two pieces (line 8), and the calls the `saxpy` task on each piece (lines 9-10).

Data parallelism is achieved in Legion by partitioning regions into *subregions*. Subregions are views onto the memory of the original parent regions. Partitions in Legion are very expressive and may subdivide regions into arbitrary subsets of elements, including overlapping, or *aliased*, subsets. Legion provides a number of partitioning operators that help users define partitions concisely [13]. For example, the `equal` operator divides a region into roughly equal subregions (line 8).

The Legion programming model permits multiple partitions of the same region (e.g., to express different access patterns) as well as replication of data across the memory hierarchy. It

```
1  @task(privileges=[RW('y') + R('x')])
2  def saxpy(S, a):
3    S.y += a * S.x
4
5  @task
6  def main():
7    S = Region([10], {'x': float32, 'y': float32})
8    P = Partition.equal(S, [2])
9    for i in IndexLaunch([2]):
10     saxpy(P[i], 1.23)
```

Listing 1. SAXPY example in Python.

```
1  struct fields {
2    x : float,
3    y : float,
4  }
5
6  task saxpy(S : region(fields), a : float)
7  where reads writes(S.y), reads(S.x) do
8    for i in S do
9      S[i].y += a * S[i].x
10   end
11 end
12
13 task main()
14   var S = region(ispace(ptr, 10), fields)
15   var P = partition(equal, S, 2)
16   for i = 0, 2 do
17     saxpy(P[i], 1.23)
18   end
19 end
```

Listing 2. SAXPY example in Regent.

does so by allowing multiple *instances* of regions or subregions (physical copies of a region's data in memory) and manages the coherence of the data in these instances based on which partition a task uses to access the region. Note that regions are allocated lazily, so that for example the region `S` at line 7 need not be allocated immediately in any particular node's memory. (In fact, for `S`, this means the region need never be allocated at all in its entirety, since it is only accessed via its subregions.) These mechanisms are largely invisible to the application, as the exact mapping from regions to instances is managed by Pygion (see Sections III-B and IV-C).

For comparison with the Pygion code in Listing 1, Listings 2 and 3 show the same example code written in Regent and C++ Legion, respectively. The Regent code in Listing 2 is mostly comparable to Pygion, and differences in line counts are mostly due to differences in syntax and formatting. The C++ code in Listing 3 on the other hand is not only substantially longer, but also exposes more low-level details of the Legion runtime system. Users of Legion in C++ must explicitly manage the IDs associated with fields (lines 1-4) and tasks (lines 6-9), must manually manage the creation of index sets (lines 32-33) and fields (lines 35-40) associated with regions, must manually set up (and as necessary serialize) the arguments to tasks (lines 53-64), must manually register tasks (not shown), and so on.

More fundamentally, the mapping between regions and instances must be manually managed in C++. Tasks that wish to obtain access to the memory associated with an instance must manually construct an *accessor* to do so (lines 14-17). Accessors are automatically managed by Pygion and Regent as described in Section III-B.

```
1  enum FIELD_IDS {
2    FID_X,
3    FID_Y,
4  };
5
6  enum TASK_IDS {
7    TID_SAXPY,
8    TID_MAIN,
9  };
10
11 void saxpy(const Task *task,
12         const std::vector<PhysicalRegion> &regions,
13         Context ctx, Runtime *runtime) {
14   FieldAccessor<READ_WRITE,float,1> acc_y(
15     regions[0], FID_Y);
16   FieldAccessor<READ_WRITE,float,1> acc_x(
17     regions[1], FID_X);
18   float a = *(const float*)(task->args);
19
20   Rect<1> rect =
21     runtime->get_index_space_domain(
22       ctx, task->regions[0].region.get_index_space());
23
24   for (PointInRectIterator<1> i(rect); i(); i++) {
25     acc_y[*i] += + a * acc_x[*i];
26   }
27 }
28
29 void main(const Task *task,
30        const std::vector<PhysicalRegion> &regions,
31        Context ctx, Runtime *runtime) {
32   IndexSpace I =
33     runtime->create_index_space(ctx, Rect<1>(0, 9));
34
35   FieldSpace F =
36     runtime->create_field_space(ctx);
37   FieldAllocator allocator =
38     runtime->create_field_allocator(ctx, F);
39   allocator.allocate_field(sizeof(float), FID_X);
40   allocator.allocate_field(sizeof(float), FID_Y);
41
42   LogicalRegion S =
43     runtime->create_logical_region(ctx, I, F);
44
45   IndexSpace colors =
46     runtime->create_index_space(ctx, Rect<1>(0, 1));
47
48   IndexPartition IP =
49     runtime->create_equal_partition(ctx, I, colors);
50   LogicalPartition P =
51     runtime->get_logical_partition(ctx, S, IP);
52
53   float a = 1.23;
54   IndexLauncher launch(
55     TID_SAXPY, colors,
56     TaskArgument((void *)&a, sizeof(a)),
57     ArgumentMap());
58   launch.add_region_requirement(RegionRequirement(
59       P, 0, READ_WRITE, EXCLUSIVE, S));
60   launch.add_region_requirement(RegionRequirement(
61       P, 0, READ_ONLY, EXCLUSIVE, S));
62   launch.add_field(0, FID_Y);
63   launch.add_field(1, FID_X);
64   runtime->execute_index_space(ctx, launch);
65 }
```

Listing 3. SAXPY example in C++.

Finally, for efficient execution by the Legion runtime, certain optimizations must be applied to the code. One of these optimizations, *index launch optimization*, is used to improve the scalability of launching tasks across many nodes by providing a concise representation of a set of tasks to be executed. This optimization has been manually applied to the C++ code (lines

53-64), but is automatically (or nearly automatically) applied in Pygion and Regent. This and other optimizations are discussed in more detail in Section IV.

Although for this particular example the Pygion and Regent code samples look quite similar, Pygion has some advantages over Regent, particularly in terms of flexibility. Regions and partitions are first-class values: both can be stored in data structures, passed to and returned from tasks, etc. Privileges are not first-class: they follow a strict stack discipline where a task can access only regions it has created itself, or ones passed as arguments (and where privileges have been declared on those arguments). To make sure that region accesses are safe (i.e., consistent with the declared privileges) and that all calls to the Legion API are well-formed, Pygion and Regent must track the subregion relationships between regions and the privileges that apply to them. In Regent, these checks must necessarily be performed statically (and must therefore be conservative). When the Regent type system cannot verify that an access is safe it must reject it at compile time, which makes it challenging or impossible to construct certain forms of data structures. By using a dynamic analysis Pygion is able to provide more flexibility while preserving the same degree of safety; the tradeoff, of course, is that errors are reported at runtime instead of at compile time. The tracking of relationships between regions and privileges is discussed in Section III.

III. Lowering Pygion to Legion

Pygion, like Regent, provides a higher-level interface than what is supported by the Legion runtime itself. This interface must be lowered to the Legion runtime to execute the program. This section describes the steps taken to lower Pygion APIs to Legion and the salient details of the implementation.

A. Tracking Regions and Privileges

As described in Section II, the subregion relationships between regions, and privileges that apply to those regions, must be tracked to ensure that region accesses are safe and that Legion API calls are well-formed.

This tracking is relatively straightforward within the body of a task: Pygion maintains a list of privileges for each region as well as a *region tree* (i.e., a tree formed by the parent-child relationships between regions and subregions). On an attempt to access the contents of a subregion, it is necessary to check that a superset of the required privileges are available, either for the subregion itself or for some ancestor in the region tree. Similarly, for a call to a subtask to be well-formed, it must identify from which ancestor region it derives privileges, which can also be determined from the region tree and privileges for each region. (An example can be seen in Listing 3 on lines 59 and 61 where `s` is provided as the 5th argument to signify that it is the ancestor which holds privileges.) These dynamic checks are simple but sufficient to capture the precise relationships between regions. In contrast, Regent must perform a type-based alias analysis which can lose precision, leading to reduced flexibility.

When passing subregions to a subtask, the list of privileges associated with each region is cleared (inside of the subtask) and replaced with the privileges associated with the subtask itself. The region tree is serialized and passed with the arguments so that the relationships between regions can be identified. (The Python `pickle` module is used for serialization, which preserves object identity within a set of objects if they are serialized at the same time.) For efficiency and to avoid passing unneeded context, the region tree is truncated at the least common ancestor among all the subregions of a given region for which the subtask has requested privileges.

Regions can be passed to subtasks inside of data structures as long as any that require privileges are also named explicitly as separate arguments. For efficiency, Pygion does not attempt to recursively scan data structures for regions, but relies on `pickle` to preserve object identity among a set of serialized objects to ensure that the regions match up correctly. This is a capability for which limited support is available in Regent, due to the need to track regions at the type system level.

If a subtask creates a new region, it can return that region to the caller, and the caller will inherit the privileges along with the region. Pygion correctly tracks the ownership of created regions, including when returned out of a subtask. Pygion also correctly tracks the region tree if a subregion is returned (or a data structure that contains a region and zero or more subregions of that region).

B. Accessors

Region instances can be organized according to any of a wide variety of data layouts: C or Fortran array order, struct-of-arrays or array-of-structs, or complex hybrid layouts (e.g., optimized for vectorization or tiling). To ensure that access to instances is efficient, Legion exposes instances as a separate type from regions (`PhysicalRegion` in C++, see line 12 of Listing 3) and requires the user to use a templated class `FieldAccessor` to access them (lines 14-17).

Pygion manages this transparently on behalf of the user. Fields of a region are exposed as NumPy arrays (via `asarray` to avoid copying), and the mapping of regions to instances is managed automatically. NumPy natively provides support for a variety of data layouts, avoiding the need for specialized code that is visible to the user.

C. Calling Convention

Legion provides the building blocks for users to construct calls to subtasks however they choose, but the exact details of the calling convention are left up to the user. For example, in Listing 3 line 56, the pass-by-value argument `a` is serialized simply by packing it into a buffer. Objects with special significance to the runtime are passed separately: regions (lines 58-63), futures (described in Section IV-B), etc. are passed to subtasks by different sets of runtime calls.

Pygion supports two calling conventions: the Regent calling convention [14], and one native to Pygion. In the Regent calling convention, arguments are packed into a struct along with a bitmask which specifies which of the arguments (if

any) are being passed via futures. Additional arrays are passed containing the field IDs of any regions contained in the arguments. Legion runtime objects such as regions are represented by handles that are safe to pass between nodes, but otherwise only plain-old-data types are supported.

The Pygion native calling convention is substantially more flexible. Arguments are serialized via `pickle`. In practice, `pickle` is a universal standard in Python, so nearly any kind of data structure can be encoded. `pickle` maintains object identity within a set of serialized objects so that large data structures are passed correctly. Certain runtime objects (such as regions) are preprocessed prior to serialization (e.g., to clear the list of existing privileges and to minimize the extent of the region tree which must be serialized). For large data structures passed repeatedly, serialization time can be amortized by storing the data structure in a future and passing this future to each task.

As noted in [14], the grouping of fields into region requirements impacts the performance of the dynamic analysis employed by the Legion runtime. For this purpose Pygion uses the same grouping algorithm as Regent to ensure optimal packing of fields into region requirements.

D. Automatic Memory Management

Python provides automatic memory management via reference counting, combined with a garbage collector designed to detect and break cycles of garbage values. This memory management strategy becomes somewhat more complicated in the presence of tasks and distributed execution: a value passed in as an argument to a subtask cannot be freed inside of the subtask even if the subtask no longer has need of it, because the parent task might still be using it. Similarly a value returned from a subtask is serialized and then (from the subtask's perspective) appears to go out of scope, even though the value itself is actually returned to caller. In both of these cases, the reference counting scheme in Python cannot be solely relied upon, because the parent and child tasks may execute on different nodes of a distributed-memory cluster, and thus the references will necessarily be broken whenever the values are serialized.

To mitigate this issue and preserve automatic memory management (i.e., not require manual deletion of Legion runtime objects), Pygion augments Python's reference counting with a notion of *ownership*, along with an escape analysis which determines when a value escapes a task.

A task that allocates a value is considered to own it, and if the value does not escape, it is deallocated via Python's normal reference counting scheme. All Legion objects are tracked by weak references, and at the end of a task, any weak references that are still valid are considered to escape. This situation can occur in one of two ways: either (a) the object is (possibly transitively) referenced from the task's return value, or (b) the object is in a cycle which has not yet been collected by the garbage collector. We consider the object to have escaped in both cases; while it would be possible to run the Python garbage collector to catch all cyclic garbage at the end of a task, this would impose too high a performance penalty. Thanks to Python's reference counting implementation, only cycles of garbage are at risk of escaping this way; otherwise collection is entirely deterministic. In our experience, referring to a Legion object from a reference cycle is not common as Legion objects do not have any user-controlled outgoing references.

Values that escape have the ownership bit set on serialization, so that the caller task takes ownership of the value (once it is deserialized). In contrast, values serialized in all other cases (such as in the arguments to a subtask call) do not have the ownership bit set (so that for example subtasks do not attempt to deallocate values owned by a parent task). Legion automatically considers any non-deallocated value to have escaped at the end of a task, so this is a good match for Legion's semantics, and it is substantially more flexible than Regent, which must do any escape analysis statically at compile time.

Note that Legion tracks objects such as regions and futures passed to subtasks, so it is not necessary to track these references in Pygion. If the parent task completes without allowing these values to escape, Pygion will instruct Legion to deallocate them, but the deallocation will be deferred by the runtime until the corresponding subtasks have completed.

IV. Optimizations

One potential concern in developing a Python interface for task-based programming is that a naive lowering of high-level constructs to the lower-level runtime interface is known to be far from optimal [7]. We show that most of Regent's optimizations for task-based programs can be provided in Python using dynamic analysis, in combination with careful API design. Out of seven optimizations presented in [7], five can be performed automatically or nearly automatically, one cannot be performed, and one has been rendered irrelevant by unrelated improvements in the underlying runtime infrastructure. An additional optimization reported in [8] has been replaced with a dynamic counterpart described in [15]. In this section we briefly describe the optimizations, their design and implementation in Pygion, and for the one optimization that cannot be performed, suggest a way in which it might potentially be implemented in the future.

A. Index Launches

Index launches are a construct that enable the runtime analysis for a set of N tasks to be performed in $\mathcal{O}(1)$ time instead of $\mathcal{O}(N)$ by leveraging a concise, $\mathcal{O}(1)$-space description of the tasks to be executed. (Note that $\mathcal{O}(1)$-time execution additionally requires the use of control replication (Section IV-D); otherwise launches are executed with an $\mathcal{O}(\log N)$ broadcast tree.) Therefore this is an optimization that is critical to the scalable execution of dynamic task-based code.

An index launch as understood by the Legion runtime consists of a *launch domain*, a task (to be instantiated once for each point in the domain), and arguments (pass-by-value, regions, futures, etc.). An example of a C++ index launch can be seen in Listing 3 lines 53-64. Region arguments to

```
1 @task
2 def main():
3   S = Region([10], {'x': float32, 'y': float32})
4   P = Partition.equal(S, [2])
5   index_launch([2], saxpy, P[ID], 1.23)
```

Listing 4. SAXPY example with constant time launches.

the launch are specified as a *projection*, or function from a point i in the launch domain to the particular subregion that is the argument to the ith task in the launch. In general this can take the form $\lambda i.P[f(i)]$ where f is any function and P is a partition, but by far the most common projection is the identity $\lambda i.P[i]$. In lines 59 and 61 the argument `0` specifies the identity projection on the partition argument `P`.

The challenge in developing an index launch optimization for Python is how to capture the projections of region arguments for the launch. We achieve this through a combination of careful API design along with symbolic execution of region expressions. In Pygion, `IndexLaunch` is a special iterator which records the series of task calls issued while the iterator is active (see Listing 1 line 9). The loop variable `i` is a symbolic value, which can be coerced to a concrete value, but also can be used with region expressions such as `P[i]` to generate a projection expression (line 10). These expressions are understood by the index launch implementation and generate the appropriate Legion calls in the backend.

Because Pygion performs this optimization dynamically, there are limitations to the impact that it has. In particular, while the optimization successfully reduces runtime analysis cost to $\mathcal{O}(1)$, the loop must still execute $\mathcal{O}(N)$ times (and $\mathcal{O}(N)$ storage is required) because the task arguments are not generally known to be loop invariant or projections. To reduce both the time and space complexity of packing arguments to $\mathcal{O}(1)$, Pygion provides a second form of index launch, called *constant time launch*.

This form of index launch can be seen in line 5 of Listing 4. In the sample, `P[ID]` is a symbolic expression representing the projection $\lambda i.P[i]$ (i.e., `ID` is the implicit loop variable). Arguments are encoded only once, and projections such as `P[ID]` are resolved in a post-processing pass over the arguments. Because the post-processing pass occurs in parallel (within the spawned tasks themselves), it costs $\mathcal{O}(N/M)$ where M is the number of processors used to execute the launch, and in practice is effectively constant time.

B. Futures

Subtasks run asynchronously with respect to the caller. To avoid prematurely blocking on the result of a subtask, which would serialize execution, task calls return *futures* which represent the yet-to-be-computed results. In Legion C++, futures are exposed to the user, and must be manually passed through a separate set of runtime APIs to be passed to other subtasks. In Pygion this is mostly transparent, and futures passed to tasks are automatically added via the appropriate runtime APIs (and deserialized appropriately in argument post-processing).

```
1 @task
2 def main():
3   S = Region([10], {'x': float32, 'y': float32})
4   T = Region([10], {'x': float32, 'y': float32})
5   saxpy(S, 1.23)
6   saxpy(T, 4.56)
7   print(S.x)
8   print(T.x)
```

Listing 5. Example with suboptimal mapping under conservative Legion runtime assumptions which is optimized by Pygion.

Although the differences between futures and concrete values can mostly be hidden in idiomatic Pygion code that uses duck typing [16], programs that explicitly check the types of task return values will be able to observe that the values are futures. This is a necessary tradeoff due to the lack of a fully static analysis and optimization capability in Pygion. In contrast, Regent programs cannot observe whether the future optimization is enabled or disabled, as the Regent compiler prevents any differences from being visible by the user.

C. Mapping

Region data can be stored in one or more instances as described in Section II. Though Legion automatically manages the coherence of instances, the conservative assumptions made by the runtime by default can result in suboptimal performance when an application performs a series of repeated task calls.

Regions are automatically *mapped* to instances at the start of a task, and must be *unmapped* to avoid data races with subtasks that have conflicting privileges on the same data. For example, consider a parent and child task that both have read-write access to a region: sequential semantics requires that the parent must unmap the region before the call and then subsequently map the region again after the call (blocking in the map operation to synchronize on the completion of the subtask), otherwise data races would be possible due to concurrent, read-write access to the same data by both the parent and child tasks. By default Legion assumes such races are possible and automatically inserts the required map and unmap calls to force the parent to wait for the completion of the child task; but this is suboptimal in the common case where a task repeatedly launches subtasks without any intervening accesses to the data.

Listing 5 shows an example which is suboptimal under the Legion runtime's conservative assumptions. By default, Legion inserts map and unmap calls around each task call, causing the main task to block on both lines 5 and 6, even though these tasks are otherwise non-interfering and the main task does not actually attempt to access the data until lines 7-8. Thus without further optimization, this program actually does not achieve parallel execution at all. In C++ the user must manually insert map and unmap calls to avoid this behavior. For example, if the example in Listing 5 were written in C++, the user might choose to unmap both `S` and `T` before line 5 and map both after line 6. Regent and Pygion automatically perform this optimization to avoid premature blocking.

Pygion automatically optimizes mapping by tracking the liveness of the NumPy arrays that wrap the fields of instances

(again, thanks to Python's deterministic reference counting implementation and with the same caveats regarding cyclic garbage). Mapping and unmapping is performed lazily, at the point where a subtask is launched (if the last data access was local) or where a local data access is made (if the last operation was a subtask). This strategy provides improved precision in the case of conditionals, compared to Regent's flow-sensitive analysis which must consider all possible execution paths.

D. Optimizations Performed Externally

The following optimizations provided by Regent are also available in Pygion, but are provided directly by the Legion runtime or another dependency.

Because the fields of regions in Pygion are exposed as NumPy arrays, pointer check elision and vectorization are performed by NumPy. In most cases, users who write idiomatic Legion code use bulk NumPy operations, which are generally amenable to amortizing any necessary checks such as pointer checks. (Pointers are really offsets into arrays in Legion, so pointer checks are subsumed into NumPy's existing bounds checks.) Similarly, NumPy provides vectorized implementations of these bulk operations, making a separate vectorization pass unnecessary. For cases where NumPy implementations might benefit from other optimizations such as loop fusion and/or tiling, Numba [17] can be used to generate these high-performance implementations. Or Regent itself can be called from Pygion, enabling automatic generation of efficient CPU and GPU implementations.

Control replication [8] is an optimization that substantially improves the scalability of a Legion program by converting the repeated fork-join style parallelism of index launches into efficient, SPMD-style code.

In normal execution, the main task is executed on one node. This node can become a scalability bottleneck as the subtasks of the main task must be launched from the same node the main task is running on (though they may execute on other nodes). Under control replicated execution, the main task is instead executed on all nodes simultaneously in SPMD fashion. The Legion runtime filters the set of tasks executed by each node so that each task is only executed once. The Legion runtime also automatically inserts data movement and synchronization to preserve the original sequential semantics of the program. In this way the SPMD nature of the execution is not visible to the user as long as the main task is deterministic. Therefore control replication avoids a sequential bottleneck because the analysis and execution of tasks is distributed across the machine.

Control replication was first implemented in the Regent compiler but is now directly implemented in the Legion runtime [15]. In the following experiments we use exclusively the Legion implementation of this optimization.

E. Optimizations No Longer Necessary

The dynamic branch elision optimization in Regent, which improves the performance of certain access patterns where data might be located in any of a set of regions, is no longer necessary in recent versions of the Legion runtime. The runtime now provides support for *co-location constraints* on tasks which require Legion to place the constrained regions into a single instance together, eliminating the need for any dynamic checks.

F. Optimizations Not Performed

An important optimization in the Legion runtime is the ability to designate tasks as *leaf* or *inner*. Leaf tasks access data locally but do not launch subtasks. Inner tasks launch subtasks but do not directly access data. Knowing that a task is inner means no instances need to be mapped for a task (and therefore the task can begin to execute even before the data is ready); knowing that a task is leaf means that several important runtime tests become cheaper because the task cannot launch subtasks.

Regent analyzes the body of each task to automatically determine these designations, but Pygion's use of interpreted Python makes it challenging to apply static analysis to the bodies of tasks (and dynamic analysis is insufficient to determine these designations, as the best one could do would be to abort the program if the designation were violated).

As a workaround, users can manually annotate tasks in Pygion as leaf and/or inner by way of optional keyword arguments to the `@task` decorator (not shown in the code samples). The cost of adding this annotation to tasks is low, but it does require users to be familiar with the definitions of leaf and inner to make the correct annotations. (Though note that if the user makes a mistake, an error will be reported at runtime and will not be permitted to corrupt the runtime state of the program.)

A potential future approach to implementing this optimization could rely on speculation support in the Legion runtime. Similar to optimistic concurrency schemes, Legion provides the ability to speculate on what properties a task might have (in this case leaf and/or inner), and to roll back the execution in the case that the speculated property does not hold. This capability comes at a cost (in particular, additional copies of data must be made in memory or on disk to ensure rollback is possible in the event of a missed speculation), but it seems likely that in long-running iterative applications, the speculation would be likely to converge quickly, reducing the cost of these mechanisms. Our experience indicates that tasks do not dynamically switch between leaf and inner, so this approach is likely to work well.

V. Evaluation

We present an evaluation of Pygion on up to 512 nodes of the Piz Daint supercomputer [10]. Piz Daint is a Cray XC50 machine with one Intel Xeon E5-2690 v3 (12 physical cores) and one NVIDIA Tesla P100 per node. We use the system default installations of GCC 6.2.0 and CUDA 9.1.85. Legion uses GASNet-EX 2019.3.0 as its communication layer [18]. Regent uses LLVM 3.8.1 for code generation [9]. Pygion uses Python 3.7.3, NumPy 1.16.4, and CFFI 1.12.3.

We consider three already-optimized Regent applications, and versions of each application where the main task has been ported to Pygion. The applications include: Stencil, a 9-point, star-shaped stencil on a grid [19]; Circuit, an electrical

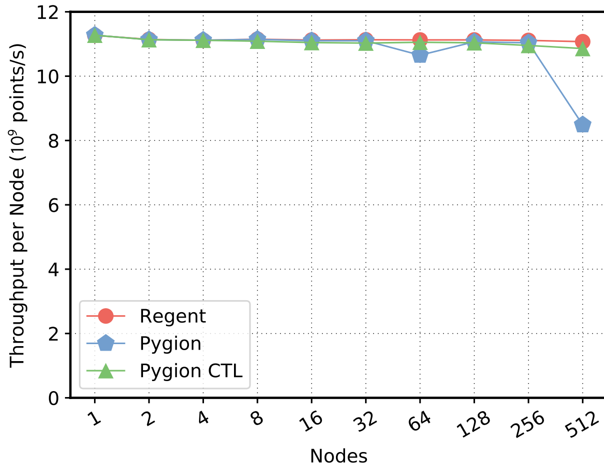

Fig. 1. Stencil weak scaling, 9×10^8 points/node.

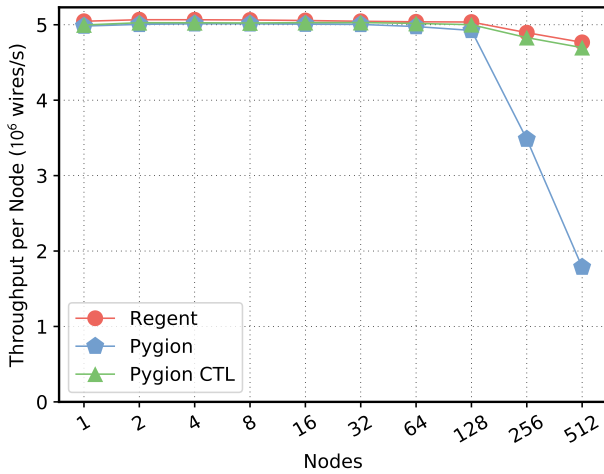

Fig. 2. Circuit weak scaling, 2×10^5 wires/node.

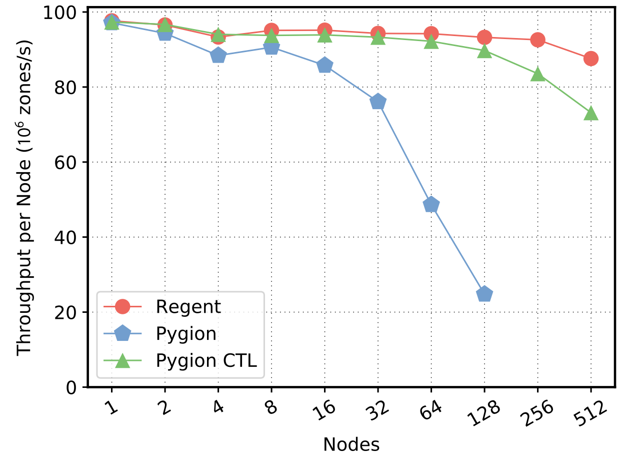

Fig. 3. Pennant weak scaling, 7.4×10^6 zones/node.

circuit simulation on an unstructured graph [8]; and Pennant, a Lagrangian hydrodynamics simulation on a 2D unstructured mesh [20]. In order to maintain as much of an apples-to-apples comparison as possible, we reuse the original Regent implementations of the tasks in each application (aside from the main task); this allows us to make use of Regent's high-performance CUDA code generator to target the GPUs on Piz Daint.

To demonstrate the scalability of Pygion, we conduct weak scaling experiments on Piz Daint up to 512 nodes. The results are presented in Figures 1, 2 and 3. For each application, we consider three versions: Regent (the baseline), Pygion, and Pygion with constant time launches (CTL). Each data point in the graphs is the average of 5 runs. We use the following problem sizes: 9×10^8 points/node for Stencil, 2×10^5 wires/node for Circuit, and 7.4×10^6 zones/node for Pennant. The applications have been configured to run 50, 50, and 30 time steps, respectively.

Thanks to the use of the existing high-performance kernels, Pygion achieves performance parity at small node counts for all codes (within 1% in absolute performance). This is not surprising, as the main task executes asynchronously from the rest of the application, and therefore does not impact performance as long as the average throughput (in tasks per second launched by the main task) exceeds the rate at which the machine can execute them.

Pygion CTL achieves weak scaling parallel efficiency of 96%, 94% and 75% respectively for Stencil, Circuit, and Pennant (vs. 98%, 94% and 90% for Regent) at 512 nodes. Without CTL, the scalability of Pygion is limited, as the $\mathcal{O}(N)$ time complexity of the packing of index launch arguments grows to dominate execution time at large node counts. This effect is most visible in Figure 3, as Pennant has the largest number of tasks per iteration of the time-step loop (and those tasks are of relatively small granularity), as well as a global reduction to compute dt for the next time step (which prevents the Legion runtime from analyzing tasks more than one iteration ahead of the actual execution of the program).

Although the asymptotic factors are important, constant factors also matter. Notably, Regent does *not* use constant time launches, and so also incurs $\mathcal{O}(N)$ time complexity in the packing of arguments, but with a constant factor that is so much smaller that it is not an issue up to 512 nodes. The reduction in asymptotic complexity is much more important for Pygion because the constant factors on packing arguments are so much higher.

For two of the three application (Stencil and Circuit), Pygion's scalable execution with CTL ensures that the applications achieve performance parity across all node counts (within 2% in absolute performance). Pennant begins to drop at 128 nodes, and achieves within 16% of Regent's performance at 512 nodes. At the time of writing, we are currently investigating the cause of the Pennant performance degradation.

VI. Related Work

Among the existing task-based programming systems for high-performance computing, by far the most common languages used for programming are C and C++. Legion [3],

PaRSEC (with dynamic task discovery) [4], and StarPU [5] support distributed-memory, while OpenMP (as of version 4.0) [21], OmpSs [22] and Kokkos [23] run on shared-memory systems. These systems share a number of common features: tasks appear to execute in program order, dependencies between tasks are determined by the arguments supplied to task calls along with the privileges requested by tasks, and tasks can be offloaded to available GPUs (with data movement managed by the system). Among these systems, Legion is the only one that provides support for partitioning [13]; the others require users to explicitly reorganize data in applications that use multiple access patterns. However, in general the use of C/C++ represents a significant barrier for scientists not already familiar with traditional HPC programming models and languages.

PyCOMPSs [24] and Dask [25] provide support for task-based programming in Python. Like Pygion, the use of Python in these systems improves usability for scientists not already familiar with lower-level programming languages such as C/C++. However, these systems lack features of Legion, such as control replication, without which performance and scalability can be limited [15]. As with most of the systems above, PyCOMPSs and Dask also lack support for partitioning.

An alternative approach explored in PaRSEC (with parameterized task graphs) [26] is to provide a domain-specific language (DSL) which can generate task graphs automatically. In this approach a DSL compiler reads a program representation (in the case of PaRSEC, a recursive, algebraic description of a task graph) and generates code to execute the tasks described in the program. These approaches can improve usability within a domain, as long as the target programs are well supported by the domain-specific semantics.

On the other hand, some languages focus on more general-purpose support for task-based programming. This is the case for Regent [7], a language which directly targets the Legion programming model. Regent provides a model which is higher-level than the Legion C++ interface, and the Regent compiler translates programs into efficient code for the Legion runtime. The Regent type system is also richer than C++, and directly tracks various program properties to ensure correct usage of the model [6]. However these type restrictions also limit the flexibility of the Regent language and make certain programming patterns difficult.

The Sequoia language [27] for array-based programs offers a form of task-based parallelism where tasks are automatically, recursively decomposed for optimized execution on deep memory hierarchies. Various optimizations in the Sequoia compiler ensure very high performance [28]. However, the compiler requires intimate knowledge of the program, including the sizes of all input arrays and the exact configuration of the target machine, to be available at compile time in order to apply these optimizations. This approach makes it impractical to apply to more dynamic problems.

Domain-specific programming frameworks such as Uintah [29] provide support for constructing directed acyclic graphs (DAGs) of tasks that can be executed asynchronously. Such models can provide improved programmability by applying domain-specific assumptions, and therefore are not applicable outside of the chosen domain.

In contrast to the implicitly parallel systems above, certain task-based runtimes provide explicitly parallel program semantics. In OCR [30] and Realm [31], the DAG of tasks is explicitly specified by the user, instead of being inferred via a static or dynamic analysis of the arguments to tasks. These systems are typically intended to be used by library and framework authors rather than directly by end-users, as the code to construct DAGs of tasks can be verbose and error-prone.

Others systems aim to directly improve the usability of explicit parallelism. These include partitioned global address space (PGAS) languages Chapel [32], Fortran coarrays [33], Titanium [34], UPC [35], and X10 [36]. Although the details vary, the common elements include the ability to hold references to (and possibly directly access) the contents of memory on remote machines and in many cases the ability to launch tasks to execute locally or remotely. Alternatively, actor models such as Charm++ [37] ensure that no such remote reference are held, and instead data movement and synchronization occurs via message passing between objects. However, as these systems are explicitly parallel they typically do not prevent all of the possible pitfalls that can occur with parallel programming.

At the other extreme, Legate [15] and Dask [25] provide support for running unmodified or minimally-edited NumPy [12] programs on distributed machines. Though NumPy itself is entirely sequential, these approaches work because most NumPy operations work in bulk, over an entire array, and can often be executed lazily (and therefore asynchronously). Internally, Legate is based on Legion, whereas Dask builds on its own task-based programming model. These approaches are appealing because they minimize the amount of work required to understand the programming model, but they rely on heuristics that may not provide optimal performance for any given problem. In cases where the heuristics fail, the user may be forced to turn to other programming models that more directly support the parallelism required.

VII. Conclusion

A growing population of users in the physical sciences and data analysis require supercomputers to process the ever larger data sets in these disciplines, but are unfamiliar with traditional high-performance programming models and languages. To meet the needs of these users, we have presented Pygion, a Python-based interface for the Legion task-based programming system. By leveraging the flexibility and the dynamic nature of the Python programming language, we have been able to implement an interface with expressiveness which is comparable to Regent, a dedicated language for the Legion programming model. For five out of seven optimizations presented in [7], we have shown that dynamic program analysis in Pygion is sufficient to support automatic or nearly automatic optimization in Pygion, one optimization is no longer necessary, and only one necessary optimization is intractable under this approach and requires manual user annotation of tasks.

With constant time launches, Pygion is able to achieve weak scalability that is comparable to already-optimized Regent GPU implementations on up to 512 nodes of the Piz Daint supercomputer. By reusing the original task implementations from the Regent code (except for the main task), Pygion is able to target Piz Daint's GPUs with minimal additional effort and with high performance.

Overall, these results point in a promising direction: contrary to what one might expect, Python is fast enough to be useful for writing the main tasks in non-trivial mini-apps relevant to high-performance scientific simulation. The additional flexibility afforded by the use of dynamic program analysis, and the streamlined interface, have the potential to make this a much more productive interface for writing task-based programs on modern heterogeneous supercomputers, especially for a large class of scientific users who are not familiar with traditional programming models and languages.

Acknowledgment

This material is based upon work supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID d80.

References

[1] "Linac coherent light source," https://lcls.slac.stanford.edu/, 2009.

[2] "Slac, berkeley lab researchers prepare for scientific computing on the exascale," https://www6.slac.stanford.edu/news/2016-11-03-slac-berkeley-lab-researchers-prepare-scientific-computing-exascale.aspx, 2016.

[3] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Supercomputing (SC)*, 2012.

[4] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in parsec: A data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '17. New York, NY, USA: ACM, 2017, pp. 6:1–6:8. [Online]. Available: http://doi.acm.org/10.1145/3148226.3148233

[5] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," Inria, Tech. Rep., 2016.

[6] S. Treichler, M. Bauer, and A. Aiken, "Language support for dynamic, hierarchical data partitioning," in *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[7] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A high-productivity programming language for HPC with logical regions," in *Supercomputing (SC)*, 2015.

[8] E. Slaughter, W. Lee, S. Treichler, W. Zhang, M. Bauer, G. Shipman, P. McCormick, and A. Aiken, "Control Replication: Compiling implicit parallelism to efficient SPMD with logical regions," in *Supercomputing (SC)*, 2017.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization (CGO)*, 2004.

[10] "Piz Daint & Piz Dora - CSCS," http://www.cscs.ch/computers/piz_daint, 2016.

[11] W. McKinney, "Data structures for statistical computing in Python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 51 – 56.

[12] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, p. 22, 2011.

[13] S. Treichler, M. Bauer, R. Sharma, E. Slaughter, and A. Aiken, "Dependent partitioning," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2016, pp. 344–358.

[14] E. Slaughter, "Regent: A high-productivity programming language for implicit parallelism with logical regions," Ph.D. dissertation, Stanford University, 2017.

[15] M. Bauer and M. Garland, "Legate: Accelerated and distributed NumPy," in *Supercomputing (SC)*, 2019.

[16] "Duck typing," https://docs.python.org/3/glossary.html#term-duck-typing.

[17] "Numba," https://numba.pydata.org/, 2012.

[18] D. Bonachea and P. H. Hargrove, "GASNet-EX: A high-performance, portable communication library for exascale," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-2001174, October 2018, languages and Compilers for Parallel Computing (LCPC'18). [Online]. Available: https://escholarship.org/uc/item/0xg7b704

[19] R. F. Van der Wijngaart and T. G. Mattson, "The Parallel Research Kernels," in *HPEC*, 2014, pp. 1–6.

[20] C. R. Ferenbaugh, "PENNANT: an unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, 2014.

[21] "OpenMP application program interface," http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf, 2013.

[22] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[23] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *Extreme Scaling Workshop (XSW), 2013*, Aug 2013, pp. 18–24.

[24] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "PyCOMPSs: Parallel computational workflows in Python," *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017.

[25] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Python in Science Conference (SciPy)*, no. 130-136. Citeseer, 2015.

[26] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[27] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *SC*, November 2006.

[28] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan, "Compilation for explicitly managed memory hierarchies," in *Principles and Practice of Parallel Programming (PPoPP)*, 2007, pp. 226–236.

[29] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Investigating applications portability with the Uintah DAG-based runtime system on petascale supercomputers," in *Supercomputing (SC)*, 2013, pp. 1–12.

[30] "The Open Community Runtime interface," https://xstack.exascale-tech.com/git/public?p=ocr.git;a=blob;f=ocr/spec/ocr-1.1.0.pdf, 2014.

[31] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[32] B. L. Chamberlain, "Chapel," in *Programming Models for Parallel Computing*, P. Balaji, Ed. MIT Press, 2015, pp. 129–159.

[33] "Fortran 2008," https://wg5-fortran.org/f2008.html, 2008.

[34] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, and P. Colella, "Titanium: A high-performance Java dialect," *Concurrency Practice and Experience*, vol. 10, no. 11-13, pp. 825–836, 1998.

[35] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," UC Berkeley Technical Report: CCS-TR-99-157, 1999.

[36] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA*, 2005.

[37] L. V. Kalé and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *OOPSLA*, 1993, pp. 91–108.

Appendix A
Artifact Description

Summary of the Experiments Reported

We perform weak scaling experiments on up to 512 nodes of the Piz Daint supercomputer [10]. For reproducibility, the exact version of Legion used in the experiments has been saved in a branch, along with all scripts used to build and run. Instructions for building and running are included in the links below.

Artifact Availability

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

List of URLs and/or DOIs where artifacts are available: Project repository: https://github.com/StanfordLegion/legion/tree/papers/pygion-paw19

Instructions: https://github.com/StanfordLegion/legion/blob/papers/pygion-paw19/language/paw19_scripts/README.md

Baseline Experimental Setup, and Modifications Made for the Paper

Relevant Hardware Details: Piz Daint (Cray XC50, Intel Xeon E5-2690 v3, NVIDIA Tesla P100, Aries interconnect)

Operating Systems and Versions: CNL based on SLES 12 SP3 running Linux kernel 4.4.162

Compilers and Versions: GCC 6.2.0, CUDA 9.1.85, LLVM 3.8.1 (Regent only), Python 3.7.3 (Legion Python only)

Applications and Versions: All benchmarks are included in the Legion repository

Libraries and Versions: NumPy 1.16.4, CFFI 1.12.3

Key Algorithms: N/A

Input Datasets and Versions: N/A

Paper Modifications: N/A

Output from scripts that gathers execution environment information:

```
CRAY_CUDATOOLKIT_VERSION=9.1.85_3.18-6.0.7.0_5.1__g2eb7c52
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_mic_knl=160
PE_SMA_DEFAULT_PKGCONFIG_VARIABLES=PE_SMA_COMPFLAG_@prgenv@
PE_LIBSCI_VOLATILE_PRGENV=CRAY GNU INTEL
KSH_AUTOLOAD=1
MODULE_VERSION_STACK=3.2.10.6
LESSKEY=/etc/lesskey.bin
PE_TPSL_DEFAULT_GENCOMPS_INTEL_x86_skylake=160
PE_PETSC_DEFAULT_GENCOMPS_CRAY_skylake=86
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_sandybridge=8.6
PE_PAPI_DEFAULT_ACCEL_FAMILY_LIBS_nvidia=,-lcupti,-lcudart,-lcuda
GNU_VERSION=6.2.0
PE_MPICH_GENCOMPILERS_PGI=15.3
PE_CXX_PKGCONFIG_LIBS=mpichcxx
NNTPSERVER=news
MANPATH=/opt/nvidia/cudatoolkit9.1/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52/
      doc/man:/opt/cray/pe/perftools/7.0.3/man:/opt/cray/pe/papi
      /5.6.0.3/share/pdoc/man:/opt/cray/pe/atp/2.1.2/man:/opt/cray/
      alps/6.6.43-6.0.7.1_5.45__ga796da32.ari/man:/opt/cray/job
      /2.2.3-6.0.7.1_5.43__g6c4e934.ari/man:/opt/cray/pe/pmi/5.0.14/
      man:/opt/cray/pe/libsci/18.07.1/man:/opt/cray/pe/man/csmlversion
      :/opt/cray/pe/craype/2.5.15/man:/opt/gcc/6.2.0/snos/share/man:/
      opt/slurm/17.11.12.cscs/share/man:/opt/cray/pe/mpt/7.7.2/gni/man
      /mpich:/opt/cray/pe/modules/3.2.10.6/share/man:/opt/slurm/
      default/share/man:/usr/local/man:/usr/share/man:/opt/cray/share/
      man:/opt/cray/pe/man
SLURM_JOB_NAME=bash
PE_HDF5_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH
XALT_ETC_DIR=/apps/daint/UES/xalt/0.7.6/etc
PE_TRILINOS_DEFAULT_GENCOMPS_CRAY_x86_64=86
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_interlagos=160
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_mic_knl=16.0
PE_LIBSCI_ACC_DEFAULT_PKGCONFIG_VARIABLES=
      PE_LIBSCI_ACC_DEFAULT_NV_SUFFIX_@accelerator@
PE_FFTW_DEFAULT_TARGET_mic_knl=mic_knl
CRAY_UDREG_INCLUDE_OPTS=-I/opt/cray/udreg/2.3.2-6.0.7.1_5.13
      __g5196236.ari/include
PE_TRILINOS_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/trilinos
      /12.12.1.0/@PRGENV@/@PE_TRILINOS_DEFAULT_GENCOMPS@/
      @PE_TRILINOS_DEFAULT_TARGET@/lib/pkgconfig
PE_SMA_DEFAULT_COMPFLAG_GNU=-fcray-pointer
PE_PARALLEL_NETCDF_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/
      parallel-netcdf/1.8.1.3/@PRGENV@/
      @PE_PARALLEL_NETCDF_DEFAULT_GENCOMPS@/lib/pkgconfig
PE_NETCDF_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/netcdf
      /4.6.1.2/@PRGENV@/@PE_NETCDF_DEFAULT_GENCOMPS@/lib/pkgconfig
LIBRARYMODULES=acml:alps:cray-dwarf:cray-fftw:cray-ga:cray-hdf5:cray-
      hdf5-parallel:cray-libsci:cray-libsci_acc:cray-mpich:cray-mpich2
      :cray-mpich-abi:cray-netcdf:cray-netcdf-hdf5parallel:cray-
      parallel-netcdf:cray-petsc:cray-petsc-complex:cray-shmem:cray-
      tpsl:cray-trilinos:cudatoolkit:fftw:ga:hdf5:hdf5-parallel:iobuf:
      libfast:netcdf:netcdf-hdf5parallel:ntk:onesided:papi:petsc:petsc
      -complex:pmi:tpsl:trilinos:xt-libsci:xt-mpich2:xt-mpt:xt-papi
CRAY_SITE_LIST_DIR=/etc/opt/cray/pe/modules
XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.6
PE_SMA_DEFAULT_COMPFLAG=
PE_MPICH_ALTERNATE_LIBS_dpm=_dpm
PE_HDF5_DEFAULT_GENCOMPILERS_GNU=7.1 6.1 5.3 4.9
PE_ENV=GNU
SLURM_NODE_ALIASES=(null)
PKGCONFIG_ENABLED=1
PE_TPSL_DEFAULT_GENCOMPS_CRAY_x86_skylake=86
HOST=daint105
TERM=xterm-256color
SHELL=/usr/local/bin/bash
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_x86_skylake=7.1 6.1
PE_PETSC_DEFAULT_GENCOMPS_CRAY_sandybridge=86
PROFILEREAD=true
HISTSIZE=
SLURM_JOB_QOS=normal
PE_TRILINOS_DEFAULT_VOLATILE_PRGENV=CRAY GNU INTEL
PE_TPSL_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH:PE_LIBSCI
PE_TPSL_DEFAULT_GENCOMPS_GNU_sandybridge=71 53 49
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_x86_skylake=160
PE_PETSC_DEFAULT_GENCOMPS_INTEL_haswell=160
PE_PETSC_DEFAULT_GENCOMPS_GNU_haswell=71 53 49
PE_PARALLEL_NETCDF_DEFAULT_VOLATILE_PRGENV=GNU
PE_NETCDF_DEFAULT_VOLATILE_PRGENV=GNU
CRAY_XPMEM_POST_LINK_OPTS=-L/opt/cray/xpmem/2.2.15-6.0.7.1_5.11
      __g7549d06.ari/lib64
CRAY_UGNI_POST_LINK_OPTS=-L/opt/cray/ugni/6.0.14.0-6.0.7.1_3.13
      __gea11d3d.ari/lib64
CRAYPE_DIR=/opt/cray/pe/craype/2.5.15
SLURM_CSCS=yes
PE_MPICH_DIR_PGI_DEFAULT64=64
PE_PETSC_DEFAULT_GENCOMPS_CRAY_interlagos=86
PE_NETCDF_HDF5PARALLEL_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/
      netcdf-hdf5parallel/4.6.1.2/@PRGENV@/
      @PE_NETCDF_HDF5PARALLEL_DEFAULT_GENCOMPS@/lib/pkgconfig
PE_HDF5_PARALLEL_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/hdf5-
      parallel/1.10.2.0/@PRGENV@/@PE_HDF5_PARALLEL_DEFAULT_GENCOMPS@/
      lib/pkgconfig
PE_HDF5_DEFAULT_VOLATILE_PRGENV=GNU
PE_FFTW_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/fftw/3.3.6.5/
      @PE_FFTW_DEFAULT_TARGET@/lib/pkgconfig
ALT_LINKER=/apps/daint/UES/xalt/0.7.6/bin/ld
CRAY_MPICH2_DIR=/opt/cray/pe/mpt/7.7.2/gni/mpich-gnu/7.1
PERL5LIB=/opt/slurm/17.11.12.cscs//lib/perl5/site_perl/5.18.2/x86_64-
      linux-thread-multi:/opt/slurm/default/lib/perl5/site_perl
      /5.18.2/x86_64-linux-thread-multi:
CRAY_CUDATOOLKIT_POST_LINK_OPTS=-L/opt/nvidia/cudatoolkit9.1/9.1.85_3
      .18-6.0.7.0_5.1__g2eb7c52/lib64 -L/opt/nvidia/cudatoolkit9
      .1/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52/extras/CUPTI/lib64 -Wl,--as
      -needed -Wl,-lcupti -Wl,-lcudart -Wl,--no-as-needed -L/opt/cray/
      nvidia/default/lib64 -lcuda
PE_TPSL_DEFAULT_GENCOMPS_CRAY_mic_knl=86
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_interlagos=8.6
PE_LIBSCI_DEFAULT_GENCOMPS_GNU_x86_64=71 61 51 49
PE_GA_DEFAULT_VOLATILE_PRGENV=GNU
PE_TPSL_DEFAULT_GENCOMPS_INTEL_x86_64=160
PE_MPICH_DEFAULT_GENCOMPILERS_GNU=7.1 5.1 4.9
PE_LIBSCI_ACC_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH:PE_LIBSCI
PE_LIBSCI_ACC_DEFAULT_GENCOMPS_CRAY_x86_64=85
PERFTOOLS_VERSION=7.0.3
PE_MPICH_GENCOMPS_GNU=71 51 49
PE_PKGCONFIG_PRODUCTS=PE_LIBSCI:PE_MPICH
FPATH=:/opt/cray/pe/modules/3.2.10.6/init/sh_funcs/no_redirect:/opt/
      cray/pe/modules/3.2.10.6/init/sh_funcs/no_redirect
MORE=-sl
PE_TPSL_64_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/tpsl/18.06.1/
      @PRGENV@64/@PE_TPSL_64_DEFAULT_GENCOMPS@/
      @PE_TPSL_64_DEFAULT_TARGET@/lib/pkgconfig
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_haswell=86
PE_PETSC_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH:PE_LIBSCI:
      PE_HDF5_PARALLEL:PE_TPSL
```

```
PE_PAPI_DEFAULT_ACCEL_LIBS_nvidia35=,-lcupti,-lcudart,-lcuda
PE_TRILINOS_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.6
PE_CRAY_DEFAULT_FIXED_PKGCONFIG_PATH=/opt/cray/pe/parallel-netcdf
      /1.8.1.3/CRAY/8.6/lib/pkgconfig:/opt/cray/pe/netcdf-hdf5parallel
      /4.6.1.2/CRAY/8.6/lib/pkgconfig:/opt/cray/pe/netcdf/4.6.1.2/CRAY
      /8.6/lib/pkgconfig:/opt/cray/pe/hdf5-parallel/1.10.2.0/CRAY/8.6/
      lib/pkgconfig:/opt/cray/pe/hdf5/1.10.2.0/CRAY/8.6/lib/pkgconfig
      :/opt/cray/pe/ga/5.3.0.8/CRAY/8.6/lib/pkgconfig
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_sandybridge=8.6
PE_PETSC_DEFAULT_GENCOMPS_CRAY_x86_64=86
PE_LIBSCI_DEFAULT_OMP_REQUIRES_openmp=_mp
PE_FORTRAN_PKGCONFIG_LIBS=mpichf90
SLURM_SPANK_SHIFTER_GID=31707
PE_SMA_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/mpt/7.7.2/gni/
      sma@PE_SMA_DEFAULT_DIR_DEFAULT64@/lib64/pkgconfig
CRAYPAT_LD_LIBRARY_PATH=/opt/cray/pe/gcc-libs:/opt/cray/gcc-libs:/opt
      /cray/pe/perftools/7.0.3/lib64
CRAYPAT_ALPS_COMPONENT=/opt/cray/pe/perftools/7.0.3/sbin/pat_alps
ALLINEA_QUEUE_DLL=/opt/cray/pe/mpt/7.7.2/gni/mpich-gnu/7.1/lib/
      libtvmpich.so.3.0.1
PE_TRILINOS_DEFAULT_GENCOMPS_INTEL_x86_64=160
PE_LIBSCI_ACC_DEFAULT_VOLATILE_PRGENV=CRAY GNU
CRAY_MPICH_BASEDIR=/opt/cray/pe/mpt/7.7.2/gni
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_haswell=160
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_x86_skylake=86
PE_NETCDF_HDF5PARALLEL_DEFAULT_GENCOMPILERS_GNU=7.1 6.1 5.3 4.9
PE_HDF5_PARALLEL_DEFAULT_GENCOMPILERS_GNU=7.1 6.1 5.3 4.9
HISTFILESIZE=
JRE_HOME=/usr/lib64/jvm/java/jre
SLURM_NNODES=1
CRAYPE_LINK_TYPE=dynamic
PE_TRILINOS_DEFAULT_GENCOMPILERS_INTEL_x86_64=160
PE_TRILINOS_DEFAULT_GENCOMPILERS_GNU_x86_64=71 53 49
PE_TPSL_DEFAULT_GENCOMPS_CRAY_x86_64=86
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_mic_knl=16.0
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_interlagos=16.0
PE_LIBSCI_DEFAULT_VOLATILE_PRGENV=CRAY GNU INTEL
PE_FFTW_DEFAULT_TARGET_interlagos=interlagos
LD_LIBRARY_PATH=/opt/cray/pe/papi/5.6.0.3/lib64:/opt/cray/job
      /2.2.3-6.0.7.1_5.43__g6c4e934.ari/lib64:/opt/gcc/6.2.0/snos/
      lib64
LS_COLORS=no=00:fi=00:di=01;34:ln=00;36:pi=40;33:so=01;35:do=01;35:bd
      =40;33;01:cd=40;33;01:or=41;33;01:ex=00;32:*.cmd=00;32:*.exe
      =01;32:*.com=01;32:*.bat=01;32:*.btm=01;32:*.dll=01;32:*.tar
      =00;31:*.tbz=00;31:*.tgz=00;31:*.rpm=00;31:*.deb=00;31:*.arj
      =00;31:*.taz=00;31:*.lzh=00;31:*.lzma=00;31:*.zip=00;31:*.zoo
      =00;31:*.z=00;31:*.Z=00;31:*.gz=00;31:*.bz2=00;31:*.tb2=00;31:*.
      tz2=00;31:*.tbz2=00;31:*.xz=00;31:*.avi=01;35:*.bmp=01;35:*.fli
      =01;35:*.gif=01;35:*.jpg=01;35:*.jpeg=01;35:*.mng=01;35:*.mov
      =01;35:*.mpg=01;35:*.pcx=01;35:*.pbm=01;35:*.pgm=01;35:*.png
      =01;35:*.ppm=01;35:*.tga=01;35:*.tif=01;35:*.xbm=01;35:*.xpm
      =01;35:*.dl=01;35:*.gl=01;35:*.wmv=01;35:*.aiff=00;32:*.au
      =00;32:*.mid=00;32:*.mp3=00;32:*.ogg=00;32:*.voc=00;32:*.wav
      =00;32:
SLURM_LOG_ACTIONS=yes
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_haswell=16.0
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_sandybridge=7.1 5.3 4.9
PE_PETSC_DEFAULT_VOLATILE_PRGENV=CRAY CRAY64 GNU GNU64 INTEL INTEL64
PE_LIBSCI_PKGCONFIG_VARIABLES=PE_LIBSCI_OMP_REQUIRES_@openmp@:
      PE_SCI_EXT_LIBPATH:PE_SCI_EXT_LIBNAME
CRAY_RCA_POST_LINK_OPTS=-L/opt/cray/rca/2.2.18-6.0.7.1_5.47__g2aa4f39
      .ari/lib64 -lrca
PE_MPICH_FIXED_PRGENV=INTEL
PE_PKGCONFIG_LIBS=cray-cudatoolkit:AtpSigHandler:cray-rca:libsci_mpi:
      libsci:mpich
SINFO_FORMAT=%9P %5a %8s %.10l %.6c %.6z %.7D %10T %N
PE_TPSL_DEFAULT_GENCOMPS_GNU_haswell=71 53 49
PE_PETSC_DEFAULT_GENCOMPS_INTEL_sandybridge=160
PE_PETSC_DEFAULT_GENCOMPS_INTEL_interlagos=160
PE_PETSC_DEFAULT_GENCOMPS_GNU_sandybridge=71 53 49
PE_PETSC_DEFAULT_GENCOMPS_GNU_interlagos=71 53 49
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_skylake=16.0
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.6
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_mic_knl=8.6
XNLSPATH=/usr/share/X11/nls
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_sandybridge=160
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_interlagos=7.1 5.3 4.9
PE_PETSC_DEFAULT_GENCOMPS_INTEL_mic_knl=160
PE_PETSC_DEFAULT_GENCOMPS_GNU_mic_knl=53
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_haswell=8.6
PE_PAPI_DEFAULT_PKGCONFIG_VARIABLES=PE_PAPI_ACCEL_LIBS_@accelerator@
PE_LIBSCI_DEFAULT_GENCOMPS_CRAY_x86_64=86
MPICH_ABORT_ON_ERROR=1
MPICH_DIR=/opt/cray/pe/mpt/7.7.2/gni/mpich-gnu/7.1
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_haswell=8.6
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_sandybridge=16.0
PE_NETCDF_HDF5PARALLEL_DEFAULT_REQUIRED_PRODUCTS=PE_HDF5_PARALLEL
PE_HDF5_PARALLEL_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH
PE_FFTW_DEFAULT_TARGET_sandybridge=sandybridge
PE_FFTW_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH
CRAY_PRGENVGNU=loaded
ATP_POST_LINK_OPTS=-Wl,-L/opt/cray/pe/atp/2.1.2/libApp/
PE_MPICH_FORTRAN_PKGCONFIG_LIBS=mpichf90
HOSTTYPE=x86_64
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_mic_knl=5.3
RCLOCAL_PRGENV=true
TMOUT=259200
gcc_already_loaded=0
PE_TPSL_DEFAULT_GENCOMPS_GNU_interlagos=71 53 49
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.6
PE_LIBSCI_GENCOMPS_INTEL_x86_64=160
PE_LIBSCI_DEFAULT_GENCOMPILERS_INTEL_x86_64=16.0
OFFLOAD_INIT=on_start
GCC_VERSION=6.2.0
CHPL_CG_CPP_LINES=1
PE_PRODUCT_LIST=CRAY_RCA:CRAY_ALPS:DVS:CRAY_XPMEM:CRAY_DMAPP:CRAY_PMI
      :CRAY_UGNI:CRAY_UDREG:CRAY_LIBSCI:CRAYPE:CRAYPE_HASWELL:GNU:GCC:
      PERFTOOLS:CRAYPAT
FROM_HEADER=
APPS=/apps/daint
PE_TPSL_DEFAULT_GENCOMPS_GNU_x86_skylake=71 61
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_x86_64=7.1 5.3 4.9
PE_MPICH_DEFAULT_GENCOMPS_PGI=153
CRAY_MPICH_ROOTDIR=/opt/cray/pe/mpt/7.7.2
PAGER=less
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_x86_64=16.0
PE_PETSC_DEFAULT_GENCOMPS_INTEL_skylake=160
PE_PETSC_DEFAULT_GENCOMPS_GNU_skylake=61
PE_LIBSCI_GENCOMPILERS_GNU_x86_64=7.1 6.1 5.1 4.9
PE_MPICH_MODULE_NAME=cray-mpich
PE_MPICH_GENCOMPILERS_CRAY=8.6
CSHEDIT=emacs
PE_TPSL_DEFAULT_GENCOMPS_CRAY_sandybridge=86
PE_TPSL_DEFAULT_GENCOMPS_CRAY_haswell=86
PE_TPSL_64_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH:PE_LIBSCI
PE_MPICH_TARGET_VAR_nvidia20=-lcudart
PE_MPICH_DEFAULT_VOLATILE_PRGENV=CRAY GNU PGI
PE_LIBSCI_GENCOMPS_CRAY_x86_64=86
PE_LIBSCI_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.6
CRAYPAT_ROOT=/opt/cray/pe/perftools/7.0.3
XDG_CONFIG_DIRS=/etc/xdg
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_x86_64=71 53 49
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_mic_knl=71 53
PE_PARALLEL_NETCDF_DEFAULT_GENCOMPS_GNU=51 49
PE_NETCDF_DEFAULT_GENCOMPS_GNU=
PE_LIBSCI_PKGCONFIG_LIBS=libsci_mpi:libsci
PE_LIBSCI_ACC_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/libsci_acc
      /18.07.1/@PRGENV@/@PE_LIBSCI_ACC_DEFAULT_GENCOMPS@/
      @PE_LIBSCI_ACC_DEFAULT_TARGET@/lib/pkgconfig
DVS_VERSION=0.9.0
CRAY_LIBSCI_DIR=/opt/cray/pe/libsci/18.07.1
CRAY_LIBSCI_BASE_DIR=/opt/cray/pe/libsci/18.07.1
CRAY_DMAPP_INCLUDE_OPTS=-I/opt/cray/dmapp/7.1.1-6.0.7.1_5.45
      __g5a674e0.ari/include -I/opt/cray/gni-headers/5.0.12.0-6.0.7.1
      _3.11__g3b1768f.ari/include
USERMODULES=acml:alps:apprentice:apprentice2:atp:blcr:cce:chapel:cray
      -ccdb:cray-fftw:cray-ga:cray-hdf5:cray-hdf5-parallel:cray-lgdb:
      cray-libsci:cray-libsci_acc:cray-mpich:cray-mpich2:cray-mpich-
      compat:cray-netcdf:cray-netcdf-hdf5parallel:cray-parallel-netcdf
      :craypat:craype:cray-petsc:cray-petsc-complex:craypkg-gen:cray-
      shmem:cray-snplauncher:cray-tpsl:cray-trilinos:cudatoolkit:ddt:
      fftw:ga:gcc:hdf5:hdf5-parallel:intel:iobuf:java:lgdb:libfast:
      libsci_acc:mpich1:netcdf:netcdf-hdf5parallel:netcdf-nofsync:
      netcdf-nofsync-hdf5parallel:ntk:onesided:papi:parallel-netcdf:
      pathscale:perftools:perftools-lite:petsc:petsc-complex:pgi:pmi:
      PrgEnv-cray:PrgEnv-gnu:PrgEnv-intel:PrgEnv-pathscale:PrgEnv-pgi:
      stat:totalview:tpsl:trilinos:xt-asyncpe:xt-craypat:xt-lgdb:xt-
      libsci:xt-mpich2:xt-mpt:xt-papi:xt-shmem:xt-totalview
LIBGL_DEBUG=quiet
MINICOM=-c on
PE_TPSL_DEFAULT_GENCOMPS_CRAY_interlagos=86
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_x86_64=7.1 5.3 4.9
PE_PKGCONFIG_DEFAULT_PRODUCTS=PE_TRILINOS:PE_TPSL_64:PE_TPSL:PE_PETSC
      :PE_PARALLEL_NETCDF:PE_NETCDF_HDF5PARALLEL:PE_NETCDF:PE_MPICH:
      PE_LIBSCI_ACC:PE_LIBSCI:PE_HDF5_PARALLEL:PE_HDF5:PE_GA:PE_FFTW
PE_HDF5_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/hdf5/1.10.2.0/
      @PRGENV@/@PE_HDF5_DEFAULT_GENCOMPS@/lib/pkgconfig
PAT_REPORT_PRUNE_NAME=_cray$mt_execute_,_cray$mt_start_,__cray_hwpc_,
      f_cray_hwpc_,cstart,__pat_,pat_region_,PAT_,OMP.slave_loop,
      slave_entry,_new_slave_entry,_thread_pool_slave_entry,
      THREAD_POOL_join,__libc_start_main,_start,__start,start_thread,
      __wrap_,UPC_ADIO_,_upc_,upc_,__caf_,__pgas_,syscall,
      __device_stub
PE_MPICH_GENCOMPILERS_GNU=7.1 5.1 4.9
MODULE_VERSION=3.2.10.6
SLURM_TASKS_PER_NODE=24
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_haswell=7.1 5.3 4.9
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_x86_skylake=7.1 6.1
PE_PETSC_DEFAULT_GENCOMPS_CRAY_mic_knl=86
PE_PARALLEL_NETCDF_DEFAULT_GENCOMPILERS_GNU=5.1 4.9
PE_NETCDF_DEFAULT_GENCOMPILERS_GNU=7.1 6.1 5.3 4.9
PE_MPICH_DEFAULT_DIR_PGI_DEFAULT64=64
PE_FFTW_DEFAULT_TARGET_abudhabi=abudhabi
ATP_IGNORE_SIGTERM=1
XTPE_NETWORK_TARGET=aries
CSCS_CUSTOM_ENV=true
CPU=x86_64
_=/usr/bin/env
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_x86_skylake=8.6
PE_SMA_DEFAULT_DIR_CRAY_DEFAULT64=64
PE_NETCDF_HDF5PARALLEL_DEFAULT_GENCOMPS_GNU=
PE_NETCDF_HDF5PARALLEL_DEFAULT_FIXED_PRGENV=CRAY PGI INTEL
PE_HDF5_PARALLEL_DEFAULT_GENCOMPS_GNU=
PE_HDF5_PARALLEL_DEFAULT_FIXED_PRGENV=CRAY PGI INTEL
SQUEUE_SORT=-t,e,S
JAVA_BINDIR=/usr/lib64/jvm/java/bin
SLURM_JOB_ID=14250172
PE_TPSL_DEFAULT_GENCOMPS_INTEL_interlagos=160
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_mic_knl=8.6
```

```
PE_TPSL_64_DEFAULT_VOLATILE_PRGENV=CRAY CRAY64 GNU GNU64 INTEL
      INTEL64
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_sandybridge=86
CRAY_UDREG_POST_LINK_OPTS=-L/opt/cray/udreg/2.3.2-6.0.7.1_5.13
      __g5196236.ari/lib64
PE_TPSL_DEFAULT_GENCOMPS_GNU_mic_knl=71 53
CRAY_ALPS_POST_LINK_OPTS=-L/opt/cray/alps/6.6.43-6.0.7.1_5.45
      __ga796da32.ari/lib64
CRAYPE_VERSION=2.5.15
PE_MPICH_VOLATILE_PRGENV=CRAY GNU PGI
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_haswell=16.0
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_sandybridge=7.1 5.3 4.9
PE_MPICH_DEFAULT_GENCOMPS_CRAY=86
PE_LIBSCI_DEFAULT_OMP_REQUIRES=
XALT_TRANSMISSION_STYLE=directdb
_LMFILES_=/opt/cray/pe/modulefiles/modules/3.2.10.6:/opt/cray/pe/
      modulefiles/cray-mpich/7.7.2:/opt/modulefiles/slurm/17.11.12.
      cscs-1:/apps/daint/UES/easybuild/modulefiles/xalt/daint
      -2016.11:/apps/daint/UES/easybuild/modulefiles/daint-gpu:/opt/
      modulefiles/gcc/6.2.0:/opt/cray/pe/craype/2.5.15/modulefiles/
      craype-haswell:/opt/cray/pe/craype/2.5.15/modulefiles/craype-
      network-aries:/opt/cray/pe/modulefiles/craype/2.5.15:/opt/cray/
      pe/modulefiles/cray-libsci/18.07.1:/opt/cray/ari/modulefiles/
      udreg/2.3.2-6.0.7.1_5.13__g5196236.ari:/opt/cray/ari/modulefiles
      /ugni/6.0.14.0-6.0.7.1_3.13__gea11d3d.ari:/opt/cray/pe/
      modulefiles/pmi/5.0.14:/opt/cray/ari/modulefiles/dmapp
      /7.1.1-6.0.7.1_5.45__g5a674e0.ari:/opt/cray/ari/modulefiles/gni-
      headers/5.0.12.0-6.0.7.1_3.11__g3b1768f.ari:/opt/cray/ari/
      modulefiles/xpmem/2.2.15-6.0.7.1_5.11__g7549d06.ari:/opt/cray/
      ari/modulefiles/job/2.2.3-6.0.7.1_5.43__g6c4e934.ari:/opt/cray/
      ari/modulefiles/dvs/2.7_2.2.118-6.0.7.1_10.2__g58b37a2:/opt/cray
      /ari/modulefiles/alps/6.6.43-6.0.7.1_5.45__ga796da32.ari:/opt/
      cray/ari/modulefiles/rca/2.2.18-6.0.7.1_5.47__g2aa4f39.ari:/opt/
      cray/pe/modulefiles/atp/2.1.2:/opt/cray/pe/modulefiles/perftools
      -base/7.0.3:/opt/cray/pe/modulefiles/PrgEnv-gnu/6.0.4:/opt/cray/
      modulefiles/cudatoolkit/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52
TARGETMODULES=craype-abudhabi:craype-abudhabi-cu:craype-accel-host:
      craype-accel-nvidia20:craype-accel-nvidia30:craype-accel-
      nvidia35:craype-barcelona:craype-broadwell:craype-haswell:craype
      -hugepages128K:craype-hugepages128M:craype-hugepages16M:craype-
      hugepages256M:craype-hugepages2M:craype-hugepages32M:craype-
      hugepages4M:craype-hugepages512K:craype-hugepages512M:craype-
      hugepages64M:craype-hugepages8M:craype-intel-knc:craype-
      interlagos:craype-interlagos-cu:craype-istanbul:craype-ivybridge
      :craype-mc12:craype-mc8:craype-mic-knl:craype-network-aries:
      craype-network-gemini:craype-network-infiniband:craype-network-
      none:craype-network-seastar:craype-sandybridge:craype-shanghai:
      craype-target-compute_node:craype-target-local_host:craype-
      target-native:craype-xeon:xtpe-barcelona:xtpe-interlagos:xtpe-
      interlagos-cu:xtpe-istanbul:xtpe-mc12:xtpe-mc8:xtpe-network-
      gemini:xtpe-network-seastar:xtpe-shanghai:xtpe-target-native:
      xtpe-xeon
JAVA_HOME=/usr/lib64/jvm/java
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_mic_knl=7.1 5.3
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_interlagos=8.6
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_skylake=8.6
PE_LIBSCI_MODULE_NAME=cray-libsci/18.07.1
PE_LIBSCI_ACC_DEFAULT_NV_SUFFIX_nvidia20=nv20
EDITOR=emacs --no-window-system
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_x86_skylake=71 61
PE_INTEL_FIXED_PKGCONFIG_PATH=/opt/cray/pe/mpt/7.7.2/gni/mpich-intel
      /16.0/lib/pkgconfig
LANG=en_US.UTF-8
PE_MPICH_NV_LIBS_nvidia20=-lcudart
PE_LIBSCI_GENCOMPILERS_CRAY_x86_64=8.6
PE_MPICH_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/mpt/7.7.2/gni/mpich-
      @PRGENV@@PE_MPICH_DIR_DEFAULT64@/@PE_MPICH_GENCOMPS@/lib/
      pkgconfig
MODULEPATH=/opt/cray/pe/perftools/7.0.3/modulefiles:/opt/cray/pe/
      craype/2.5.15/modulefiles:/apps/daint/UES/jenkins/6.0.UP07/gpu/
      easybuild/tools/modules/all:/apps/daint/UES/jenkins/6.0.UP07/gpu
      /easybuild/modules/all:/apps/daint/modulefiles:/apps/daint/
      system/modulefiles:/apps/daint/UES/easybuild/modulefiles:/apps/
      common/UES/modulefiles:/apps/common/system/modulefiles:/opt/cray
      /pe/modulefiles:/opt/cray/modulefiles:/opt/modulefiles:/opt/cray
      /ari/modulefiles:/opt/cray/pe/ari/modulefiles
PYTHONSTARTUP=/etc/pythonstart
SHMEM_ABORT_ON_ERROR=1
LOADEDMODULES=modules/3.2.10.6:cray-mpich/7.7.2:slurm/17.11.12.cscs
      -1:xalt/daint-2016.11:daint-gpu:gcc/6.2.0:craype-haswell:craype-
      network-aries:craype/2.5.15:cray-libsci/18.07.1:udreg
      /2.3.2-6.0.7.1_5.13__g5196236.ari:ugni/6.0.14.0-6.0.7.1_3.13
      __gea11d3d.ari:pmi/5.0.14:dmapp/7.1.1-6.0.7.1_5.45__g5a674e0.ari
      :gni-headers/5.0.12.0-6.0.7.1_3.11__g3b1768f.ari:xpmem
      /2.2.15-6.0.7.1_5.11__g7549d06.ari:job/2.2.3-6.0.7.1_5.43
      __g6c4e934.ari:dvs/2.7_2.2.118-6.0.7.1_10.2__g58b37a2:alps
      /6.6.43-6.0.7.1_5.45__ga796da32.ari:rca/2.2.18-6.0.7.1_5.47
      __g2aa4f39.ari:atp/2.1.2:perftools-base/7.0.3:PrgEnv-gnu/6.0.4:
      cudatoolkit/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52
TZ=Europe/Zurich
SDK_HOME=/usr/lib64/jvm/java
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_mic_knl=16.0
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_interlagos=71 53 49
PE_PKG_CONFIG_PATH=/opt/cray/pe/cti/1.0.7/lib/pkgconfig:/opt/cray/pe/
      cti/1.0.6/lib/pkgconfig:/opt/cray/pe/cti/1.0.4/lib/pkgconfig
PE_FFTW_DEFAULT_TARGET_x86_skylake=x86_skylake
PE_FFTW_DEFAULT_TARGET_share=share
PE_FFTW_DEFAULT_TARGET_ivybridge=ivybridge
CRAY_DMAPP_POST_LINK_OPTS=-L/opt/cray/dmapp/7.1.1-6.0.7.1_5.45
      __g5a674e0.ari/lib64
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_x86_skylake=8.6
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_skylake=6.1
PE_LIBSCI_OMP_REQUIRES_openmp=_mp
PAT_BUILD_PAPI_BASEDIR=/opt/cray/pe/papi/5.6.0.3
CRAY_RCA_INCLUDE_OPTS=-I/opt/cray/rca/2.2.18-6.0.7.1_5.47__g2aa4f39.
      ari/include -I/opt/cray/krca/2.2.4-6.0.7.1_5.43__g8505b97.ari/
      include -I/opt/cray-hss-devel/8.0.0/include
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_x86_64=16.0
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_mic_knl=86
PE_MPICH_CXX_PKGCONFIG_LIBS=mpichcxx
CRAY_MPICH_DIR=/opt/cray/pe/mpt/7.7.2/gni/mpich-gnu/7.1
PE_MPICH_PKGCONFIG_VARIABLES=PE_MPICH_NV_LIBS_@accelerator@:
      PE_MPICH_ALTERNATE_LIBS_@multithreaded@:
      PE_MPICH_ALTERNATE_LIBS_@dpm@
PE_LIBSCI_DEFAULT_GENCOMPS_INTEL_x86_64=160
PE_LIBSCI_ACC_DEFAULT_GENCOMPILERS_GNU_x86_64=4.9
SQUEUE_FORMAT=%.8i %.8u %.7a %.14j %.3t %9r %19S %.10M %.10L %.5D %.4
      C
CXX=CC
PE_TPSL_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/tpsl/18.06.1/
      @PRGENV@/@PE_TPSL_DEFAULT_GENCOMPS@/@PE_TPSL_DEFAULT_TARGET@/lib
      /pkgconfig
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_x86_skylake=16.0
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_mic_knl=8.6
PE_HDF5_DEFAULT_FIXED_PRGENV=CRAY PGI INTEL
CRAY_PMI_POST_LINK_OPTS=-L/opt/cray/pe/pmi/5.0.14/lib64
APP2_STATE=7.0.3
PE_MPICH_PKGCONFIG_LIBS=mpich
CRAY_MPICH2_VER=7.7.2
HISTCONTROL=erasedups:ignorespace
PE_PARALLEL_NETCDF_DEFAULT_FIXED_PRGENV=CRAY PGI INTEL
PE_NETCDF_DEFAULT_FIXED_PRGENV=CRAY PGI INTEL
PE_MPICH_ALTERNATE_LIBS_multithreaded=_mt
PE_LIBSCI_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/libsci/18.07.1/
      @PRGENV@/@PE_LIBSCI_GENCOMPS@/@PE_LIBSCI_TARGET@/lib/pkgconfig
PE_LIBSCI_ACC_DEFAULT_GENCOMPS_GNU_x86_64=49
PE_GA_DEFAULT_GENCOMPILERS_GNU=5.3 4.9
CUDATOOLKIT_HOME=/opt/nvidia/cudatoolkit9.1/9.1.85_3.18-6.0.7.0_5.1
      __g2eb7c52
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_haswell=71 53 49
PE_PKGCONFIG_PRODUCTS_DEFAULT=PE_PAPI
PE_NETCDF_HDF5PARALLEL_DEFAULT_VOLATILE_PRGENV=GNU
PE_MPICH_TARGET_VAR_nvidia35=-lcudart
PE_HDF5_PARALLEL_DEFAULT_VOLATILE_PRGENV=GNU
CRAY_LIBSCI_VERSION=18.07.1
QT_SYSTEM_DIR=/usr/share/desktop-data
JDK_HOME=/usr/lib64/jvm/java
SHLVL=3
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_interlagos=16.0
LESS_ADVANCED_PREPROCESSOR=no
OSTYPE=linux
PE_TPSL_DEFAULT_VOLATILE_PRGENV=CRAY CRAY64 GNU GNU64 INTEL INTEL64
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_interlagos=8.6
PE_MPICH_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/mpt/7.7.2/gni/
      mpich-@PRGENV@@PE_MPICH_DEFAULT_DIR_DEFAULT64@/
      @PE_MPICH_DEFAULT_GENCOMPS@/lib/pkgconfig
PE_LIBSCI_ACC_DEFAULT_NV_SUFFIX_nvidia60=nv60
PE_TPSL_DEFAULT_GENCOMPS_INTEL_sandybridge=160
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_interlagos=86
CRAY_PMI_INCLUDE_OPTS=-I/opt/cray/pe/pmi/5.0.14/include
LS_OPTIONS=-N --color=none -T 0
XCURSOR_THEME=DMZ
SLURM_JOB_CPUS_PER_NODE=24
SLURM_CLUSTER_NAME=daint
CRAY_CUDATOOLKIT_INCLUDE_OPTS=-I/opt/nvidia/cudatoolkit9.1/9.1.85_3
      .18-6.0.7.0_5.1__g2eb7c52/include -I/opt/nvidia/cudatoolkit9
      .1/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52/extras/CUPTI/include -I/opt
      /nvidia/cudatoolkit9.1/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52/extras/
      Debugger/include
CRAY_CUDATOOLKIT_DIR=/opt/nvidia/cudatoolkit9.1/9.1.85_3.18-6.0.7.0_5
      .1__g2eb7c52
PKG_CONFIG_PATH_DEFAULT=/opt/cray/pe/papi/5.6.0.2/lib64/pkgconfig
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_haswell=8.6
GCC_PATH=/opt/gcc/6.2.0
ATP_MRNET_COMM_PATH=/opt/cray/pe/atp/2.1.2/libexec/
      atp_mrnet_commnode_wrapper
PE_MPICH_DIR_CRAY_DEFAULT64=64
CRAYPE_NETWORK_TARGET=aries
PRGENVMODULES=PrgEnv-cray:PrgEnv-gnu:PrgEnv-intel:PrgEnv-pathscale:
      PrgEnv-pgi
WINDOWMANAGER=
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_sandybridge=16.0
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_interlagos=7.1 5.3 4.9
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_mic_knl=7.1 5.3
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_haswell=7.1 5.3 4.9
SLURM_JOB_PARTITION=normal
PE_TRILINOS_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH:PE_HDF5_PARALLEL:
      PE_NETCDF_HDF5PARALLEL:PE_LIBSCI:PE_TPSL
PE_TPSL_DEFAULT_GENCOMPS_GNU_x86_64=71 53 49
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_sandybridge=16.0
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_haswell=7.1 5.3 4.9
PE_NETCDF_DEFAULT_REQUIRED_PRODUCTS=PE_HDF5
PE_MPICH_NV_LIBS=
PE_HDF5_DEFAULT_GENCOMPS_GNU=
CRAY_LIBSCI_PREFIX_DIR=/opt/cray/pe/libsci/18.07.1/GNU/6.1/x86_64
CRAY_GNI_HEADERS_INCLUDE_OPTS=-I/opt/cray/gni-headers
      /5.0.12.0-6.0.7.1_3.11__g3b1768f.ari/include
PYTHONPATH=/apps/daint/UES/xalt/0.7.6/site:/apps/daint/UES/xalt
      /0.7.6/libexec
G_FILENAME_ENCODING=@locale,UTF-8,ISO-8859-15,CP1252
```

```
LESS=-M -I -R
MACHTYPE=x86_64-suse-linux
PE_TRILINOS_DEFAULT_GENCOMPS_GNU_x86_64=71 53 49
PE_MPICH_DEFAULT_GENCOMPILERS_CRAY=8.6
PE_LIBSCI_OMP_REQUIRES=
DMAPP_ABORT_ON_ERROR=1
PE_MPICH_GENCOMPS_CRAY=86
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_sandybridge=8.6
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_interlagos=16.0
PE_MPICH_DEFAULT_GENCOMPS_GNU=71 51 49
PE_MPICH_DEFAULT_FIXED_PRGENV=INTEL
PE_LIBSCI_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH
PE_LIBSCI_ACC_DEFAULT_NV_SUFFIX_nvidia35=nv35
PE_LIBSCI_ACC_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.5
DVS_INCLUDE_OPTS=-I/opt/cray/dvs/2.7_2.2.118-6.0.7.1_10.2__g58b37a2/
      include
TOOLMODULES=apprentice:apprentice2:atp:chapel:cray-lgdb:craypat:
      craypkg-gen:cray-snplauncher:ddt:gdb:iobuf:papi:perftools:
      perftools-lite:stat:totalview:xt-craypat:xt-lgdb:xt-papi:xt-
      totalview
XDG_DATA_DIRS=/usr/share
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_sandybridge=7.1 5.3 4.9
PE_LIBSCI_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/libsci
      /18.07.1/@PRGENV@/@PE_LIBSCI_DEFAULT_GENCOMPS@/
      @PE_LIBSCI_DEFAULT_TARGET@/lib/pkgconfig
PE_GA_DEFAULT_FIXED_PRGENV=CRAY PGI INTEL
MODULESHOME=/opt/cray/pe/modules/3.2.10.6
SLURM_JOB_NUM_NODES=1
PE_PETSC_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/petsc/3.8.4.0/
      complex/@PRGENV@/@PE_PETSC_DEFAULT_GENCOMPS@/
      @PE_PETSC_DEFAULT_TARGET@/lib/pkgconfig
PE_MPICH_NV_LIBS_nvidia35=-lcudart
PELOCAL_PRGENV=true
SLURM_TIME_FORMAT=relative
PKG_CONFIG_PATH=/opt/nvidia/cudatoolkit9.1/9.1.85_3.18-6.0.7.0_5.1
      __g2eb7c52/lib64/pkgconfig:/opt/cray/rca/2.2.18-6.0.7.1_5.47
      __g2aa4f39.ari/lib64/pkgconfig:/opt/cray/alps/6.6.43-6.0.7.1_5
      .45__ga796da32.ari/lib64/pkgconfig:/opt/cray/xpmem
      /2.2.15-6.0.7.1_5.11__g7549d06.ari/lib64/pkgconfig:/opt/cray/gni
      -headers/5.0.12.0-6.0.7.1_3.11__g3b1768f.ari/lib64/pkgconfig:/
      opt/cray/dmapp/7.1.1-6.0.7.1_5.45__g5a674e0.ari/lib64/pkgconfig
      :/opt/cray/pe/pmi/5.0.14/lib64/pkgconfig:/opt/cray/ugni
      /6.0.14.0-6.0.7.1_3.13__gea11d3d.ari/lib64/pkgconfig:/opt/cray/
      udreg/2.3.2-6.0.7.1_5.13__g5196236.ari/lib64/pkgconfig:/opt/cray
      /pe/craype/2.5.15/pkg-config:/opt/cray/pe/iobuf/2.0.8/lib/
      pkgconfig:/opt/slurm/17.11.12.cscs/lib64/pkgconfig:/opt/slurm/
      default/lib64/pkgconfig:/opt/cray/pe/atp/2.1.2/lib/pkgconfig
LESSOPEN=lessopen.sh %s
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_x86_64=160
LIBSCI_BASE_DIR=/opt/cray/pe/libsci/18.07.1
CRAYPAT_OPTS_EXECUTABLE=sbin/pat-opts
PE_TPSL_DEFAULT_GENCOMPS_INTEL_mic_knl=160
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_sandybridge=71 53 49
PE_MPICH_NV_LIBS_nvidia60=-lcudart
PE_LIBSCI_DEFAULT_PKGCONFIG_VARIABLES=
      PE_LIBSCI_DEFAULT_OMP_REQUIRES_@openmp@:PE_SCI_EXT_LIBPATH:
      PE_SCI_EXT_LIBNAME
LIBSCI_VERSION=18.07.1
INFOPATH=/opt/gcc/6.2.0/snos/share/info
CC=cc
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_x86_64=7.1 5.3 4.9
PE_PGI_DEFAULT_FIXED_PKGCONFIG_PATH=/opt/cray/pe/parallel-netcdf
      /1.8.1.3/PGI/15.3/lib/pkgconfig:/opt/cray/pe/netcdf-hdf5parallel
      /4.6.1.2/PGI/17.10/lib/pkgconfig:/opt/cray/pe/netcdf/4.6.1.2/PGI
      /17.10/lib/pkgconfig:/opt/cray/pe/hdf5-parallel/1.10.2.0/PGI
      /17.10/lib/pkgconfig:/opt/cray/pe/hdf5/1.10.2.0/PGI/17.10/lib/
      pkgconfig:/opt/cray/pe/ga/5.3.0.8/PGI/17.10/lib/pkgconfig
PE_LIBSCI_GENCOMPILERS_INTEL_x86_64=16.0
PE_FFTW_DEFAULT_TARGET_broadwell=broadwell
CRAY_ALPS_INCLUDE_OPTS=-I/opt/cray/alps/6.6.43-6.0.7.1_5.45
      __ga796da32.ari/include
CRAY_CPU_TARGET=haswell
CRAY_PRE_COMPILE_OPTS=-hnetwork=aries
XDG_RUNTIME_DIR=/run/user/23600
XTPE_LINK_TYPE=dynamic
craype_already_loaded=0
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_x86_64=86
PE_PAPI_DEFAULT_ACCELL_FAMILY_LIBS=
PE_MPICH_DEFAULT_GENCOMPILERS_PGI=15.3
PE_LIBSCI_REQUIRED_PRODUCTS=PE_MPICH
CRAY_XPMEM_INCLUDE_OPTS=-I/opt/cray/xpmem/2.2.15-6.0.7.1_5.11
      __g7549d06.ari/include
CRAY_UGNI_INCLUDE_OPTS=-I/opt/cray/ugni/6.0.14.0-6.0.7.1_3.13
      __gea11d3d.ari/include
PE_MPICH_GENCOMPS_PGI=153
PE_TPSL_DEFAULT_GENCOMPS_INTEL_haswell=160
PE_LIBSCI_GENCOMPS_GNU_x86_64=71 61 51 49
PE_LIBSCI_DEFAULT_GENCOMPILERS_GNU_x86_64=7.1 6.1 5.1 4.9
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_x86_64=16.0
PE_FFTW_DEFAULT_TARGET_x86_64=x86_64
ATP_HOME=/opt/cray/pe/atp/2.1.2
LESSCLOSE=lessclose.sh %s %s
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_x86_skylake=16.0
PE_SMA_DEFAULT_DIR_PGI_DEFAULT64=64
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_haswell=16.0
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_interlagos=7.1 5.3 4.9
PE_PAPI_DEFAULT_ACCEL_LIBS=
PE_INTEL_DEFAULT_FIXED_PKGCONFIG_PATH=/opt/cray/pe/parallel-netcdf
      /1.8.1.3/INTEL/16.0/lib/pkgconfig:/opt/cray/pe/netcdf-
      hdf5parallel/4.6.1.2/INTEL/16.0/lib/pkgconfig:/opt/cray/pe/
      netcdf/4.6.1.2/INTEL/16.0/lib/pkgconfig:/opt/cray/pe/mpt/7.7.2/
      gni/mpich-intel/16.0/lib/pkgconfig:/opt/cray/pe/hdf5-parallel
      /1.10.2.0/INTEL/16.0/lib/pkgconfig:/opt/cray/pe/hdf5/1.10.2.0/
      INTEL/16.0/lib/pkgconfig:/opt/cray/pe/ga/5.3.0.8/INTEL/18.0/lib/
      pkgconfig
PE_GA_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/ga/5.3.0.8/
      @PRGENV@/@PE_GA_DEFAULT_GENCOMPS@/lib/pkgconfig
PE_GA_DEFAULT_GENCOMPS_GNU=53 49
PE_FFTW_DEFAULT_TARGET_haswell=haswell
CRAY_LD_LIBRARY_PATH=/opt/nvidia/cudatoolkit9.1/9.1.85_3.18-6.0.7.0_5
      .1__g2eb7c52/lib64:/opt/nvidia/cudatoolkit9.1/9.1.85_3
      .18-6.0.7.0_5.1__g2eb7c52/extras/CUPTI/lib64:/opt/cray/pe/
      perftools/7.0.3/lib64:/opt/cray/rca/2.2.18-6.0.7.1_5.47
      __g2aa4f39.ari/lib64:/opt/cray/alps/6.6.43-6.0.7.1_5.45
      __ga796da32.ari/lib64:/opt/cray/xpmem/2.2.15-6.0.7.1_5.11
      __g7549d06.ari/lib64:/opt/cray/dmapp/7.1.1-6.0.7.1_5.45
      __g5a674e0.ari/lib64:/opt/cray/pe/pmi/5.0.14/lib64:/opt/cray/
      ugni/6.0.14.0-6.0.7.1_3.13__gea11d3d.ari/lib64:/opt/cray/udreg
      /2.3.2-6.0.7.1_5.13__g5196236.ari/lib64:/opt/cray/pe/libsci
      /18.07.1/GNU/6.1/x86_64/lib:/opt/cray/pe/mpt/7.7.2/gni/mpich-gnu
      /7.1/lib
G_BROKEN_FILENAMES=1
SLURM_MEM_PER_NODE=61000
PE_PETSC_DEFAULT_GENCOMPS_INTEL_x86_64=160
PE_PETSC_DEFAULT_GENCOMPS_GNU_x86_64=71 53 49
PE_PETSC_DEFAULT_GENCOMPS_CRAY_haswell=86
PE_MPICH_DEFAULT_DIR_CRAY_DEFAULT64=64
JAVA_ROOT=/usr/lib64/jvm/java
COLORTERM=1
BASH_FUNC_module%%=() {  eval '/opt/cray/pe/modules/3.2.10.6/bin/
      modulecmd bash $*'
}
+ lsb_release -a
LSB Version:    n/a
Distributor ID: SUSE
Description:    SUSE Linux Enterprise Server 12 SP3
Release:        12.3
Codename:       n/a
+ uname -a
Linux daint105 4.4.162-94.72-default #1 SMP Mon Nov 12 18:57:45 UTC
      2018 (9de753f) x86_64 x86_64 x86_64 GNU/Linux
+ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                20
On-line CPU(s) list:   0-19
Thread(s) per core:    1
Core(s) per socket:    10
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz
Stepping:              2
CPU MHz:               1200.025
CPU max MHz:           3000.0000
CPU min MHz:           1200.0000
BogoMIPS:              4600.15
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              25600K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep
      mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2
      ss ht tm pbe syscall nx pdpe1gb rdtscp lm ibrs flush_l1d
      constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
      nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor
      ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca
      sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave
      avx f16c rdrand lahf_lm abm ida arat epb invpcid_single pln pts
      dtherm ssbd ibpb stibp kaiser tpr_shadow vnmi flexpriority ept
      vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm
      xsaveopt cqm_llc cqm_occup_llc
+ cat /proc/meminfo
MemTotal:       263274152 kB
MemFree:        86758608 kB
MemAvailable:   187873960 kB
Buffers:         2570964 kB
Cached:         134770224 kB
SwapCached:            0 kB
Active:          9569164 kB
Inactive:       128684260 kB
Active(anon):     935828 kB
Inactive(anon):     7048 kB
Active(file):    8633336 kB
Inactive(file): 128677212 kB
Unevictable:    15728720 kB
Mlocked:        15728720 kB
SwapTotal:      134217724 kB
SwapFree:       134217724 kB
Dirty:               980 kB
Writeback:           112 kB
AnonPages:      16641048 kB
Mapped:          1161008 kB
Shmem:             30600 kB
```

```
Slab:           19340680 kB
SReclaimable:    1339944 kB
SUnreclaim:     18000736 kB
KernelStack:       23648 kB
PageTables:        83928 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    265854800 kB
Committed_AS:   18191864 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
HardwareCorrupted:     0 kB
AnonHugePages:  15501312 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:    11571956 kB
DirectMap2M:    234747904 kB
DirectMap1G:    24117248 kB
+ inxi -F -c0
./collect_environment.sh: line 14: inxi: command not found
+ lsblk -a
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
loop0    7:0    0  2.6G  0 loop
loop1    7:1    0 34.7G  0 loop /var/opt/cray/imps-image-binding/PE/
     squash_mounts/squashfs_vSpo3d_mount_point
loop2    7:2    0        1 loop
loop3    7:3    0        0 loop
loop4    7:4    0        0 loop
loop5    7:5    0        0 loop
loop6    7:6    0        0 loop
loop7    7:7    0        0 loop
sda      8:0    0  700G  0 disk
    sda1   8:1    0 1007K  0 part
    sda2   8:2    0    2G  0 part /boot
    sda3   8:3    0   20G  0 part
    sda4   8:4    0  256G  0 part /tmp
    sda5   8:5    0  128G  0 part [SWAP]
sdb      8:16   0  1.5T  0 disk
    sdb1   8:17   0   10G  0 part /var/crash
    sdb2   8:18   0    2G  0 part /var/mmfs
    sdb3   8:19   0  1.5T  0 part /var/opt/cray/persistent
sr0     11:0    1 1024M  0 rom
+ lsscsi -s
[0:2:0:0]    disk    DELL     PERC H730 Mini   4.29  /dev/sda    751
     GB
[0:2:1:0]    disk    DELL     PERC H730 Mini   4.29  /dev/sdb   1.64
     TB
[10:0:0:0]   cd/dvd  HL-DT-ST DVD+-RW GTA0N    A3B0  /dev/sr0
     -
+ module list
++ /opt/cray/pe/modules/3.2.10.6/bin/modulecmd bash list
Currently Loaded Modulefiles:
  1) modules/3.2.10.6
  2) cray-mpich/7.7.2
  3) slurm/17.11.12.cscs-1
  4) xalt/daint-2016.11
  5) daint-gpu
  6) gcc/6.2.0
  7) craype-haswell
  8) craype-network-aries
  9) craype/2.5.15
 10) cray-libsci/18.07.1
 11) udreg/2.3.2-6.0.7.1_5.13__g5196236.ari
 12) ugni/6.0.14.0-6.0.7.1_3.13__gea11d3d.ari
 13) pmi/5.0.14
 14) dmapp/7.1.1-6.0.7.1_5.45__g5a674e0.ari
 15) gni-headers/5.0.12.0-6.0.7.1_3.11__g3b1768f.ari
 16) xpmem/2.2.15-6.0.7.1_5.11__g7549d06.ari
 17) job/2.2.3-6.0.7.1_5.43__g6c4e934.ari
 18) dvs/2.7_2.2.118-6.0.7.1_10.2__g58b37a2
 19) alps/6.6.43-6.0.7.1_5.45__ga796da32.ari
 20) rca/2.2.18-6.0.7.1_5.47__g2aa4f39.ari
 21) atp/2.1.2
 22) perftools-base/7.0.3
 23) PrgEnv-gnu/6.0.4
 24) cudatoolkit/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52
```