NEW DIRECTIONS IN UNCERTAINTY QUANTIFICATION USING TASK-BASED PROGRAMMING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Manolis Papadakis
December 2019

Abstract

Many problems of interest in modern computational science and engineering involve multiple interacting physical processes. To properly simulate such problems computational scientists must combine multiple physics solvers in a tightly-coupled environment. Mainstream low-level HPC programming frameworks are a poor match for the complexities of such applications, especially given the need to adapt to rapidly-changing hardware platforms, that are becoming increasingly heterogeneous. Moreover, the predictive power of such complex applications is hindered by their large number of uncertain inputs, making Uncertainty Quantification (UQ) a critical feature of any multi-physics solver. However, the current practice of UQ involves a high degree of manual effort, and current tools do not take full advantage of the available parallelism.

We believe that Task-Based Programming, a distributed programming approach that has gained popularity in recent years, can significantly improve multiple aspects of building multi-physics applications. Task-based systems, with their higher level of abstraction, are simpler to program than traditional HPC frameworks like MPI, and the final code is easier to tune, port and maintain. Additionally, such systems have the potential to significantly improve the time required to perform UQ studies, not just by improving how efficiently ensembles can be executed, but also by automating many aspects of UQ that are currently performed by hand, and possibly even enable new UQ algorithms.

In this dissertation we report on our experience developing Soleil-X, a multi-physics solver supporting coupled simulations of fluid, particles and radiation, in the Legion task-based programming system. We discuss how to design and optimize

such an application, and evaluate our solver's scalability on Sierra, a leadership-class supercomputer. We implement UQ support for Soleil-X and use it to prototype a framework for automatically constructing an optimal low-fidelity model for use in UQ studies. We apply this framework on a medium-size simulation and compare its performance against human experts. Finally, we discuss the problem of optimizing the execution of UQ ensembles.

Acknowledgments

I would like to thank the following people, for their help in completing this work:

- my PhD advisor, Alex Aiken
- the PI of the PSAAP II program, Gianluca Iaccarino
- the members of my Reading and Orals Committees: Pat Hanrahan, Chris Ré and Kunle Olukotun
- from the PSAAP II CS team: Wonchan Lee and Sierra Kaplan-Nelson
- from the PSAAP II ME team: Lluís Jofre-Cruanyes, Hilario Torres, Ari Frankel and Thomas Jaravel
- from the Legion team: Elliott Slaughter, Mike Bauer, Sean Treichler, Zhihao Jia and Seema Mirchandaney
- former members of the PSAAP II team: Gilbert Bernstein, Chinmayee Shah and Tom Economon
- Pamphile Roy, for his help in setting up the case study from Chapter 4
- the participants in our user study from Chapter 4: Hilario Torres, Mario Di Renzo, Jeremy Horwitz, Zach del Rosario, Jani Adcock, Heather Pacella, Immanuvel Paul, Andrew Banko, Ji Hoon Kim, Ohi Dibua, João Reis and Gianluca Iaccarino

Contents

List of Tables

List of Figures

Chapter 1

Task-Based Programming

In this chapter we give an introduction to task-based programming, with a particular focus on Legion, the task-based programming system we used for this work.

1.1 The Legion Task-Based Programming System

Task-based programming is an asynchronous programming model where the work to be completed is decomposed in units called *tasks*. Tasks are implicitly ordered according to their dependencies, and their scheduling and execution are handled by a separate runtime component. Legion [14] is a specific example of a task-based programming model for High-Performance Computing applications that targets distributed, heterogeneous computing platforms.

Regions are the core data abstraction in Legion. A *region* is conceptually analogous to a database table. Row identifiers correspond to index points, which can be opaque pointers or (potentially multi-dimensional) points with an implied geometric relationship. The set of valid indices for a region forms the region's *index space*. Fields are the Legion equivalent of database columns, and the set of fields in a region is that region's *field space*. Regions can be partitioned into sub-regions, and this partitioning can be disjoint (the sub-regions do not overlap) or aliased. Sub-regions do not contain their own data, but instead reference the data contained in the parent region. A region can be partitioned multiple times, and sub-regions can be partitioned

recursively into finer regions.

Tasks in Legion must explicitly name the (sub-)regions they access, and what privileges they will need over each, e.g. read-only, read-write, write-discard (the task will overwrite every element in the region without reading its previous value) or reduce (the task will only perform reductions into elements of the region using a commutative and associative operator such as addition or multiplication). Multiple tasks can execute concurrently, however the code within a task executes sequentially. Tasks may recursively launch sub-tasks, but can only pass to them regions they already have access to, and only with the same or fewer privileges. Tasks can create new regions and partitions.

In addition to launching tasks and computing on region contents, application code can emit *fills*, which lazily replace the contents of a region with a specified value, and explicit *copy* operations, that copy the contents of one region to another.

Legion employs a dynamic runtime that executes concurrently with the application. This runtime analyzes tasks in the order they are launched and dynamically constructs a task graph recording tasks' data dependencies by analyzing the regions and privileges the tasks request. The runtime also automatically schedules any required data movement, based on task dependencies and the placement of tasks in the machine; for example, if task t_1 produces an output region r that is an input to task t_2, then the runtime will automatically schedule a copy if t_1 and t_2 are executed on different nodes of a parallel machine. The task graph is executed asynchronously; a task can start executing as soon as its dependencies are satisfied. The dependencies in the task graph are computed based on the program order in which tasks are launched, which implies that Legion maintains sequential execution semantics, even though tasks may ultimately execute in parallel.

The runtime analysis does not wait for launched tasks to finish; as long as the application does not itself block on the result of a task execution it can continue to launch sub-tasks for the runtime to analyze and schedule. Allowing new tasks to be launched and analyzed even while other tasks are executing ensures that as many operations as possible are in flight, thus hiding the latency of the runtime analysis with useful work. Therefore, Legion applications should generally maximize the number of

task launches performed prior to making any blocking calls. This is made easier by Legion's use of futures to encapsulate certain scalar results of task execution, which can be passed directly to other tasks, without waiting on their result.

Realm [9] is the low-level layer that actually executes the task graph on a distributed machine. Realm uses the GASNet [15] networking layer to support a variety of network substrates. Realm also provides a custom OpenMP [19] implementation and an interface to the CUDA [49] runtime, which is used to control Nvidia GPUs.

Regent [61] is a domain-specific programming language that targets the Legion runtime, and treats regions and tasks as first-class primitives. The Regent compiler is implemented using the Terra toolkit [21], itself based on the LLVM compiler framework [37]. The compiler supports various features that make Legion programs more concise and easier to write:

- It can statically check many of the requirements that the runtime would perform dynamically.
- It has extensive support for metaprogramming, using the Lua [32] high-level scripting language.
- It transparently handles futures returned by tasks, and will automatically lift many operations to work on futures, to avoid blocking as much as possible.
- It analyzes task code for potential optimizations, e.g. it detects when a task will never call any sub-tasks (a *leaf* task), or when it will not directly access the contents of any regions (an *inner* task).
- It automatically transforms loops of non-interfering task launches into *index-space launches*, a special mechanism for analyzing and launching multiple tasks as a unit, whose complexity remains constant regardless of the number of tasks.
- It can generate code for a variety of architectures (e.g. OpenMP, x86 with vector intrinsics, CUDA) from the same source.

The final component of a Legion application is the *mapper*, which is responsible for making every machine-specific choice when executing the task graph. For instance,

the mapper decides which variant of a task to use (e.g. GPU or CPU version), what node and processor it will execute on, and at what priority level. The mapper also decides where in the memory hierarchy to place the data associated with each region that a task needs to access, and how that data will be laid out (e.g. array-of-struct, AoS, or struct-of-array, SoA). This instantiation of a region into concrete memory is a *physical instance* of the region. The mapper is separate from the rest of the application, which allows programmers to adapt their execution policy to different workloads and machines without having to modify the main code. Moreover, any valid set of mapping decisions is guaranteed to produce the same answer, i.e. mapping decisions only impact performance and are orthogonal to correctness. Finally, instead of writing a custom mapper, programmers may opt to use the default mapper, which uses heuristics to make generic mapping decisions for any application.

The system diagram in Figure 1.1 outlines the high-level structure of a typical Regent-based Legion application.

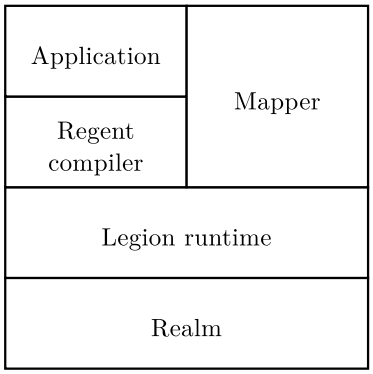

Figure 1.1: Legion system diagram

1.2 Benefits of Task-Based Programming

By choosing to build our distributed application on top of a task-based runtime like Legion instead of using a communication layer like MPI [71] directly we potentially miss out on some flexibility, and have to give up some of our computational resources to the runtime analysis. In return we get a programming system that has visibility

into how the data is stored and distributed, and what parts of it are used by each step in the computation, which is advantageous in multiple ways:

Better effort-performance tradeoff Programming directly to the communication layer allows programmers to theoretically achieve optimal performance for a given application and target machine. In practice, however, programming at this layer is so complex that reaching this optimum would require a prohibitive amount of effort, and even getting to a basic implementation takes significant time. Legion offers a better tradeoff of programming effort and performance: It is much easier to get started with a Legion implementation, since the runtime handles the correctness requirements of distributed execution, and getting to an acceptable level of performance requires a much more modest time investment.

Adaptability For a given application the optimal mapping policy is likely to be different for every machine, and even for every input. If the code is written in a low-level programming system like MPI, making changes to the data or work distribution is likely to require substantial changes to the program. In contrast, Legion's decoupling of functional correctness from mapping decisions allows users to easily adapt to different configurations, and encourages experimentation with sophisticated mapping policies (e.g. independent partitioning of different regions).

Cleaner code In an MPI application the computation, communication and mapping decisions are all interleaved within the same code. Conversely, in a Legion application the computation is separate from the mapping code, and the communication is the responsibility of the runtime. This makes Legion applications both cleaner and easier to maintain.

More parallelism The Legion runtime transparently overlaps communication and computation, schedules future work asynchronously and allows one processor to get arbitrarily ahead of the others. These features enable Legion to extract the maximum amount of parallelism from an application, with no programmer intervention.

Compositionality In Legion, modules that have been written independently can be composed by simply invoking them in the appropriate order (the runtime will automatically infer their data dependencies and schedule them safely).

Portability Legion applications typically require only minor changes to be ported onto different architectures; the application code normally stays mostly the same, and only the mapper may need to be modified.

Ease of programming When writing an application in MPI the programmer must manually reason about data dependencies and set up communication patterns around them. The programming system itself knows nothing about these dependencies, and cannot verify that the programmer's communication pattern matches them. In contrast, Legion infers all data dependencies automatically from the code, thus guaranteeing that no dependency will ever be missed, or an extraneous one introduced. Also, Legion's sequential semantics make it much easier to reason about program correctness. Finally, the Legion runtime automatically handles the error-prone aspects of distributed programming, e.g. scheduling, data movement, data coherence and synchronization.

Dynamism The Legion mapping interface gives programmers the ability to update their mapping decisions dynamically, e.g. to do work stealing, load balancing or dynamically re-partition application data based on profiling information.

We note that, to take full advantage of these features, programmers need to adapt somewhat to the separation of functionality and mapping policy. Ideally the main code of the application will not make any decisions that may affect performance, including even choices like AoS vs SoA data layout and partitioning strategy for the different regions. Instead the application should implement all potentially relevant strategies and let the mapper choose what to use, according to the available resources.

Chapter 2

Soleil-X: A Task-Based Multi-Physics Solver

In this chapter we report on our experience building a multi-physics solver, Soleil-X [69], on top of the Legion task-based programming system.

2.1 Motivation

We believe that multi-physics applications such as Soleil-X are a good match for task-based approaches, for two reasons:

- Easier development: Multi-physics solvers can take advantage of the inherent compositionality of task-based programming approaches; the different physics can be developed and tested independently, and combined by simply calling them in the appropriate order. As mentioned previously, the runtime manages a lot of the correctness requirements of distributed execution, such as data movement and synchronization, and presents a sequential-looking programming model, that allows domain experts to focus on the physics without worrying about the complexities of distributed programming.
- Easier scaling and porting: When implementing distributed applications of this

scope manually, programmers typically end up making sub-optimal implementation decisions in an effort to keep the complexity of the code under control. In contrast, task-based programming models separate the correctness of the code from most performance-relevant decisions, so applications can be optimized to specific hardware after they have been fully developed and verified, and porting them to a different machine requires only small changes that cannot compromise correctness. In addition, the multiple levels of parallelism inherent in multi-physics applications (see Section 2.3) are straightforward to model using hierarchical task launches, allowing a task-based runtime to extract all of the available parallelism much more easily than human programmers.

Building Soleil-X served as a significant exercise to gauge how well these arguments hold in practice. Due to its size and complexity, Soleil-X also served as a good benchmark application to push the limits of the Legion ecosystem. It also informed the direction of the project: As will be discussed later, codes such as Soleil-X that combine multiple solvers in a single simulation are necessary to model a wide class of interesting physical phenomena. Therefore, they represent an important class of applications that any programming system targeting computational physics should aim to support.

2.2 Preliminaries

Throughout this chapter we will use pseudocode snippets to demonstrate the general shape of different computations performed by Soleil-X. In this section we describe the format we follow in these snippets.

Most of the coding constructs we use should be familiar to programmers, with the exception of the following:

- Different fields of the same region are represented as separate named arrays, e.g. `foo[i]` represents an access to field `foo` of some region `r` on index point `i`.
- **`for`** `i` **`in`** `R` loops over the indices of region `R` in an unspecified order.

- `for i = a..b` iterates in order from `a` up to (and including) `b`.
- `a, b ~ c d` represents a dependency of values `a` and `b` on values `c` and `d`. This statement abstracts away from the actual computation performed on `c` and `d` to derive `a` and `b`.

Note that, in an effort to make our examples easy to understand, we do not show the full set of dependencies for each operation, only the most relevant ones. For example, we might use a dependency like `a[i] ~ b[i-1], b[i+1]` to abstractly represent the assignment `a[i] = c[i] * (b[i-1] + b[i+1])`, i.e. we would not show the centered access to `c[i]`, as the crux of the computation is the two-sided stencil access on `b`. Additionally, we may skip intermediate fields and group multiple fields into one.

2.3 High-Level Description

The Soleil-X solver was developed in the Regent programming language, in the context of the PSAAP II program [1] at Stanford, a collaborative effort between the Mechanical Engineering and Computer Science departments with the goal of developing an Exascale-ready computational framework for the high-fidelity predictive simulation of irradiated particle-laden turbulent flows. Such flows are encountered in a wide range of natural phenomena and industrial applications, e.g. the interaction of gases, soot, and thermal radiation in combustion systems, the coalescence and evaporation of droplets in atmospheric clouds and ocean sprays, and the operation of volumetric particle-based solar energy receivers.

Figure 2.1 shows the high-level structure of Soleil-X, and Figure 2.2 the different levels of parallelism within the solver. In the next sections we present the different modules of Soleil-X in detail. The physics behind the solver's algorithms will not be discussed in detail, but a comprehensive discussion can be found in previous work [34, 55]. For the purposes of our discussion, which focuses on the computational challenges of implementing multi-physics applications, it will be sufficient to consider an abstracted version of each computation performed by the solver.

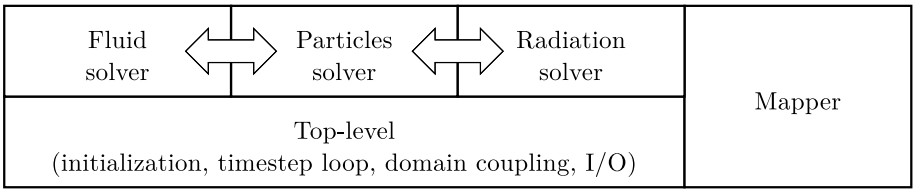

Figure 2.1: Soleil-X architecture diagram

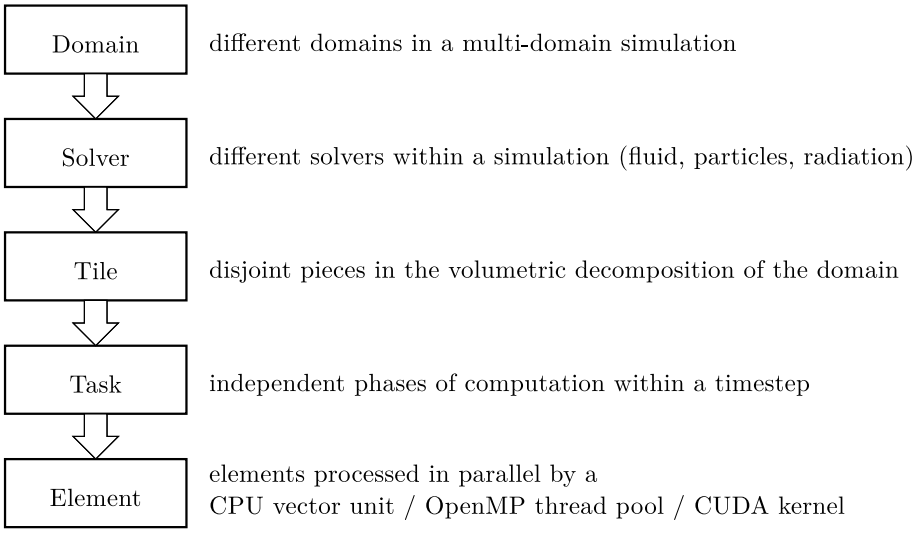

Figure 2.2: Levels of parallelism within Soleil-X

2.4 Fluid Solver

Soleil-X's fluid mechanics module is responsible for solving the variable-density compressible Navier-Stokes equations governing fluid motion [25]. The simulation follows an Eulerian formulation and a finite volume discretization scheme, using a uniform 3D Cartesian grid. Differentiation is performed using a 2nd order central finite differencing scheme. Time integration is performed using an explicit Runge-Kutta scheme; 2nd, 3rd and 4th order schemes are supported. The size of the timestep (i.e. the amount of physical time we advance the simulation on every iteration) can be set manually or computed dynamically through a CFL condition check, to the maximum safe value for the current state of the simulation. The simulation supports multiple initialization schemes (uniform, random, Taylor-Green Vortex [66]) and boundary conditions (wall, periodic, inflow, outflow). Turbulence can be artificially introduced to a domain, using an HIT (Homogeneous Isotropic Turbulence) forcing method [13].

For the purposes of this discussion it is sufficient to consider a simplified version of the flow computation, as presented in Figure 2.3. At each time step the fluid solver loops over all fluid cells, applies a stencil to read values from surrounding cells, and uses those values to update values at the center of the stencil.

To give an idea of the computational intensity (CI) of Soleil-X we analyze three tasks from the fluid solver, specifically the most computationally intensive tasks from each of the three phases of the fluid solve, as shown in Figure 2.3. We give a detailed listing of these tasks' code in Figure 2.4, along with a line-by-line breakdown of the amount of computation they perform, in terms of floating-point operations (FLOPs)[1]. We measure the CI of these tasks in Table 2.1. Since none of these tasks exhibits a CI ratio higher than 0.4 FLOPs/byte, the fluid solver is considered a memory-bound computation.

When executing on more than one processor we decompose our simulation domain volumetrically, in equally-sized tiles. This split defines our primary, disjoint partitioning of the fluid region. When executing on more than one node we perform

[1]Note that, for the purposes of this discussion, we do not differentiate between different floating-point operations, such as addition and multiplication.

```
-- Compute velocity gradients
for c in Fluid do
  dv_dx[c] ~ v[c+{-1, 0, 0}], v[c+{+1, 0, 0}]
  dv_dy[c] ~ v[c+{ 0,-1, 0}], v[c+{ 0,+1, 0}]
  dv_dz[c] ~ v[c+{ 0, 0,-1}], v[c+{ 0, 0,+1}]
-- Use velocity gradients to compute face fluxes
for c in Fluid do
  flux_x[c] ~
    v[c],          dv_dy[c],          dv_dz[c],
    v[c+{+1,0,0}], dv_dy[c+{+1,0,0}], dv_dz[c+{+1,0,0}]
  flux_y[c] ~
    v[c],          dv_dx[c],          dv_dz[c],
    v[c+{0,+1,0}], dv_dx[c+{0,+1,0}], dv_dz[c+{0,+1,0}]
  flux_z[c] ~
    v[c],          dv_dx[c],          dv_dy[c],
    v[c+{0,0,+1}], dv_dx[c+{0,0,+1}], dv_dy[c+{0,0,+1}]
-- Use face fluxes to compute change in velocity
for c in Fluid do
  dv_dt[c] ~
    flux_x[c],          flux_y[c],          flux_z[c],
    flux_x[c+{-1,0,0}], flux_y[c+{0,-1,0}], flux_z[c+{0,0,-1}]
  v[c] ~ dv_dt[c]
```

Figure 2.3: Pseudocode representation of the fluid solver's main loop. This is an abstract representation of how the solver uses the values of the velocity field `v` at time `t` to derive the velocity values at time `t+1`. See Section 2.2 for a description of our pseudocode format.

	CalcDvDx	CalcFluxesX	UpdateDrDtX
vector fields read	v	v , rv , dv_dy , dv_dz	
scalar fields read		r , rE , P , T	dr_dt , r_flux_x
vector fields written	dv_dx	rv_flux_x	
scalar fields written		r_flux_x , rE_flux_x	dr_dt
bytes accessed per element	48	168	24
FLOPs per element	6	67	2
CI ratio (FLOPs/byte)	0.125	0.3988	0.0833

Table 2.1: Computational intensity of fluid solver tasks. The amount of memory accessed per element is derived from the set of fields that the task reads and/or writes (we do not differentiate between reads and writes when computing the CI ratio). Each scalar field is a double-precision floating-point value (8 bytes). Each vector field is a tuple of three doubles (24 bytes total). To compute the FLOP count we summed the operations performed in the main loop of each task (see Figure 2.4), which iterates over every element of the fluid region.

```
task CalcDvDx(...)
  for c in Fluid do
    dv_dx[c] = (v[c+{+1,0,0}] - v[c+{-1,0,0}]) / (2 * dx)      — 6 FLOPs

task CalcFluxesX(...)
  for c in Fluid do
    var v_face = (v[c] + v[c+{1,0,0}]) / 2                     — 6 FLOPs
    var r_face = (r[c] + r[c+{1,0,0}]) / 2                     — 2 FLOPs
    var rv_face = (rv[c] + rv[c+{1,0,0}]) / 2                  — 6 FLOPs
    var rE_face = (rE[c] + rE[c+{1,0,0}]) / 2                  — 2 FLOPs
    var P_face = (P[c] + P[c+{1,0,0}]) / 2                     — 2 FLOPs
    var v_xface = (v[c+{1,0,0}] - v[c]) / dx                   — 6 FLOPs
    var T_xface = (T[c+{1,0,0}] - T[c]) / dx                   — 2 FLOPs
    var v_yface = (dv_dy[c] + dv_dy[c+{1,0,0}]) / 2            — 6 FLOPs
    var v_zface = (dv_dz[c] + dv_dz[c+{1,0,0}]) / 2            — 6 FLOPs
    var sigma = mu * {                                         — 3 FLOPs
      (4 * v_xface[0] - 2 * v_yface[1] - 2 * v_zface[2]) / 3, — 6 FLOPs
      v_xface[1] + v_yface[0],                                 — 1 FLOP
      v_xface[2] + v_zface[0]}                                 — 1 FLOP
    r_flux_x[c] = v_face[0] * r_face                           — 1 FLOP
    rv_flux_x[c] = v_face[0] * rv_face - sigma + {P_face,0,0} — 7 FLOPs
    rE_flux_x[c] = v_face[0] * (rE_face + P_face)              — 2 FLOPs
                 - dot(v_face, sigma)                          — 6 FLOPs
                 + c * T_xface                                 — 2 FLOPs

task UpdateDrDtX(...)
  for c in Fluid do
    dr_dt[c] += (r_flux_x[c+{-1,0,0}] - r_flux_x[c]) / dx      — 2 FLOPs
```

Figure 2.4: Detailed code listings of fluid solver tasks, with FLOP counts. We are only showing the main loop of the most computationally intensive task from each of the three phases of the fluid solve, as shown in Figure 2.3. Intermediate values have been folded in to reduce code size. Note that certain operations, such as the addition of velocity values, are vector operations on 3-element tuples. See Section 2.2 for a description of our pseudocode format.

a two-level split: we first split our domain volumetrically across all nodes, then further split each node's sub-domain into equal tiles, one for each processor on that node. Each task (which represents a different step of the fluid solve) is now launched multiple times, once per tile. Our mapper decides on a static allocation of tiles to processors, and ensures that task launches centered on the same tile always run on the same processor, so that the data associated with that tile never has to move.

Some of the tasks in Soleil-X need to perform a stencil around every cell. For those tasks it is not sufficient to pass just the tile corresponding to the executing processor. Instead, additional pieces of the main fluid region are needed, to cover the extra ghost cells read by the stencil near the boundaries of the tile. This problem is automatically handled by Regent's parallelizer [38], a compiler pass that analyzes the code of a task and augments it with the additional region arguments and access checks required to perform the stencil safely in a distributed setting. The parallelizer also sets up additional partitions over the original regions, as required by the new parallelized tasks. For example, when processing a task that contains the stencil access `1,0,01,0,0dv_dx[c]~v[c+-], v[c++]`, Regent's parallelizer would emit two extra region arguments, for the slices of cells directly to the left and right of the current tile. The main fluid region would also be partitioned in two additional ways, naming the left and right ghost sub-regions for each tile.

2.5 Particles Solver

Soleil-X simulates particles using a Lagrangian formulation, following the point-particle model [18, 65], whereby particles are represented as infinitesimally small points, each moving independently through the fluid.

Particles are strongly coupled with the fluid in both directions (e.g. the fluid flow affects each particle's trajectory, and each particle transfers heat to the fluid around it). Particles only need to trade information with the fluid around them, and primarily with the fluid cell that contains them. We chose to maintain this connection explicitly through a `cell` field on the particles region (essentially a pointer to the element of the fluid region that represents the cell containing the particle). Any time a particle

changes position this cell pointer may need to be updated.

Following this modeling strategy, the fluid-to-particles coupling code can be written according to the pseudocode in Figure 2.5. As can be seen from that code, to compute the gradient on a particle's velocity we need not only the velocity at the center of the particle's containing cell, but also the velocity on that cell's 27 neighbors (so we can perform trilinear interpolation), i.e. we need to perform a 27-point stencil. The reverse direction, from particles to fluid, represented abstractly in Figure 2.6, accesses just the containing cell of each particle. Notice that this code, by virtue of performing a reduction, can be executed in parallel across particles, even though different particles may need to update the same cell (that they are both contained in).

```
for p in Particles do
  var c = cell[p]
  dvp_dt[p] ~
    v[c+{-1,-1,-1}], v[c+{-1,-1, 0}], v[c+{-1,-1,+1}],
    v[c+{-1, 0,-1}], v[c+{-1, 0, 0}], v[c+{-1, 0,+1}],
    ...
    v[c+{+1,+1,-1}], v[c+{+1,+1, 0}], v[c+{+1,+1,+1}]
```

Figure 2.5: Pseudocode representation of the fluid-to-particles coupling code. See Section 2.2 for a description of our pseudocode format.

```
for p in Particles do
  var c = cell[p]
  dv_dt[c] += f( dvp_dt[p] )
```

Figure 2.6: Pseudocode representation of the particles-to-fluid coupling code. See Section 2.2 for a description of our pseudocode format.

This modeling strategy poses a problem when running on more than one processor: the fluid has been partitioned into disjoint tiles, so we need to be careful in our partitioning of the particles to make sure every time we follow a `cell` pointer the information for the corresponding cell is available on the node where the code is running. The simplest way to achieve this, and indeed the one we chose, is to "co-partition" the particles with the fluid: the particles are partitioned such that every

particle which resides in the physical space covered by a fluid tile will be assigned to the same processor as that tile. This decision can be communicated to the parallelizer through an invariant declaration, and the parallelizer will take advantage of it when parallelizing the fluid/particle coupling tasks, even producing the ghost fluid regions required for the 27-point stencil automatically.

One issue with this partitioning scheme is that particles change positions as the simulation progresses, and may eventually move to a different fluid tile. It is our responsibility to maintain the co-partitioning invariant as particles move. The obvious solution to this problem, to re-partition the particles region every time particles move, will eventually lead to sparse sub-regions as particles drift from their original positions, and we reach a point where a particle's index in the region and its physical position are uncorrelated. The Legion runtime, as of the time of this writing, is not optimized for handling sparse regions. To work around this performance limitation we chose to manually transfer particle information to the appropriate sub-region whenever a particle crosses to a different fluid tile.

This decision, in turn, required a change to our particles data model. Soleil-X needed to allow the distribution of particles across the simulated space to drift somewhat from uniform, so the number of particles per tile was allowed to not be constant. However, regions in Legion are statically-sized, so we had to leave some slack space on each particles sub-region, to accommodate for some imbalance. This meant that every index in the particles region may or may not contain an actual particle, and an additional "valid" flag had to be introduced to keep track of this information. The simulation code now had to always check this flag before operating on a particle. Also, an index in the particles region could no longer serve as a stable reference to a particle, as the underlying particle could now be moved to a different tile, invalidating that slot in the process.

Figure 2.7 shows how we move particles from one sub-region to a neighboring one, through the use of an intermediate "transfer queue" region. This queue is sized to accommodate the maximum number of moving particles we expect to see (we assume that particles are slow-moving, such that only a small percentage of them moves out of their containing tile on every timestep).

```
-- Assign slots on the transfer queue for moving particles
for s in SrcParticles do
  must_move[s] = 1 if src_valid[s] and out_of_bounds(s) else 0
var n_xfers = sum(must_move)
queue_slot = prefix_sum(must_move)
-- Copy moving particles to the transfer queue
for s in SrcParticles do
  if must_move[s] then
    var q = queue_slot[s] - 1
    queue[q] = src_payload[s]
    src_valid[s] = false
-- Number all empty slots on the destination region
for d in DstParticles do
  can_fill[d] = 0 if dst_valid[d] else 1
dst_slot = prefix_sum(can_fill)
-- Copy moving particles from the transfer queue
for d in DstParticles do
  if not dst_valid[d] then
    var q = dst_slot[d] - 1
    if q < n_xfers then
      dst_payload[d] = queue[q]
      dst_valid[d] = true
```

Figure 2.7: Pseudocode representation of the particle exchange code. We show how particles can be moved from a `SrcParticles` sub-region (shown to have only two fields, the `src_valid` flag, that marks whether a slot is filled, and the `src_payload` field, that stands in for all the information that Soleil-X tracks for every particle) to a `DstParticles` sub-region through the use of a separate `queue` region, in a way that allows for efficient GPU code generation. In the real implementation we use a separate queue for each pair of tiles that a particle can move between. See Section 2.2 for a description of our pseudocode format.

As discussed in Section 2.8, one of our goals with Soleil-X was to be able to run the entire simulation on a GPU, and that includes the particle exchange code. This code is not trivial to write for GPUs, as it needs to assign a unique slot on the transfer queue to different particles in the source sub-region, which is not a trivially parallelizable operation. The snippet in Figure 2.7 shows how we achieved this while using only constructs that can be implemented with reasonable efficiency on a GPU, such as scalar reduction and prefix sum [59].

2.6 Radiation Solver

Soleil-X's radiation solver uses the Discrete Ordinates Method (DOM) to solve the Radiative Transfer Equation (RTE), which governs the interaction of radiation with a system of fluid and particles [46]. This method is computationally expensive, but can accurately model a variety of phenomena, such as reflection, scattering, absorption, shadowing, blackbody emission and externally imposed illumination. In simple cases we can fall back to a simpler, "optically thin" model, where each particle is simply assumed to absorb the same amount of radiation. This model may be good enough for cases where particle concentration is low enough that it is safe to disregard shadowing effects.

More specifically, the DOM solver accepts a grid of concentration values and computes, by running a "sweep" step repeatedly until convergence, the amount of radiation absorbed in every cell of that grid. The value updated on every sweep step is the radiation intensity on every cell center and face of the grid, which is different for each direction in 3D space. The solver needs to discretize this set of directions into a finite set `M` of solid angles, each associated with a weight representing the area of the spherical sector it defines. The sweep for a specific angle proceeds cell-by-cell; a cell is ready to be processed if the intensities on all three entering ("upstream") faces are known. These values are used to update the cell-centered and exiting ("downstream") face intensities. Therefore, each angle must start from a corner of the grid, and "sweep" through all cells. Figure 2.8 shows an abstracted version of this process.

```
-- Update wall intensity according to reflection
for m in Angles where points_to_xneg(m) do
  I_x[0,y,m] ~ I_x[0,y,n] for n in Angles where points_to_xpos(n)
for m in Angles where points_to_xpos(m) do
  I_x[X-1,y,m] ~ I_x[X-1,y,n] for n in Angles where points_to_xneg(n)
for m in Angles where points_to_yneg(m) do
  I_y[x,0,m] ~ I_y[x,0,n] for n in Angles where points_to_ypos(n)
for m in Angles where points_to_ypos(m) do
  I_y[x,Y-1,m] ~ I_y[x,Y-1,n] for n in Angles where points_to_yneg(n)
-- Include contribution of other boundary conditions on walls
...
-- Perform sweeps across domain, to update face and cell intensity values
for m in Angles where points_to_xpos(m) and points_to_ypos(m) do
  for x = 0..X-1 do for y = 0..Y-1 do
    I[x,y,m], I_x[x+1,y,m], I_y[x,y+1,m] ~ S[x,y], I_x[x,y,m], I_y[x,y,m]
for m in Angles where points_to_xpos(m) and points_to_yneg(m) do
  for x = 0..X-1 do for y = Y-1..0 do
    I[x,y,m], I_x[x+1,y,m], I_y[x,y-1,m] ~ S[x,y], I_x[x,y,m], I_y[x,y,m]
for m in Angles where points_to_xneg(m) and points_to_ypos(m) do
  for x = X-1..0 do for y = 0..Y-1 do
    I[x,y,m], I_x[x-1,y,m], I_y[x,y+1,m] ~ S[x,y], I_x[x,y,m], I_y[x,y,m]
for m in Angles where points_to_xneg(m) and points_to_yneg(m) do
  for x = X-1..0 do for y = Y-1..0 do
    I[x,y,m], I_x[x-1,y,m], I_y[x,y-1,m] ~ S[x,y], I_x[x,y,m], I_y[x,y,m]
-- Reduce intensity field across all angles, to derive source term
S[x,y] ~ I[x,y,m] for m in Angles
```

Figure 2.8: Pseudocode representation of the DOM solver's main loop (simplified 2D version). `I[x,y,m]` represents the radiation intensity at the center of cell `(x,y)`, over the spherical sector centered around solid angle `m`. Similarly, `I_x` and `I_y` track the intensity values on the x- and y-faces of cells. The final output of the solver is derived from field `S`; this loop would be repeated until all values in this field have converged. See Section 2.2 for a description of our pseudocode format.

Because all the angles in the same quadrant will sweep the grid in the same direction, DOM implementations usually group angles into eight quadrants and perform the sweeps as eight separate computations which, by virtue of having no data dependencies, can run in parallel.

For our applications of interest it happens that the fluid is essentially transparent to radiation, and it is the particles that actually absorb energy and transfer it to the fluid as heat. Consequently, the concentration field that the DOM solver takes as input is derived exclusively from the concentration of the particles. Figure 2.9 shows how we compute the concentration field by counting the number of particles in each radiation grid cell, by means of a reduction operation (in the actual code we do not just count particles, but instead aggregate values based on the particles' physical properties). Note that we chose not to have an additional radiation cell pointer on particles. Instead we assumed, based on experience with similar solvers, that the radiation grid would only ever be at most as fine as the fluid grid, and probably coarser. If we additionally constrain this coarsening factor to be an integer, then every fluid cell would be fully contained inside exactly one radiation cell. We can then take advantage of this relationship, to decompose the link from particle to containing radiation cell into first following the particle's `cell` pointer to the containing fluid cell, then moving from that to the containing radiation cell. Given this coupling scheme, it is natural to partition the radiation grid following the partitioning of the fluid and particles regions. After the DOM solver has converged, we use the same coupling scheme to distribute the radiation intensity on each radiation cell back onto the particles it contains.

```
for p in particles do
  var c = cell[p]
  var r = rad_cell[c]
  concentration[r] += 1
```

Figure 2.9: Pseudocode representation of the radiation grid initialization code. We first follow the `cell` pointer of a particle to find its containing fluid cell, retrieve the radiation cell which contains that, and reduce into the radiation cell's concentration field. See Section 2.2 for a description of our pseudocode format

In the case of multi-processor executions the sweep pattern will also occur at the tile level. The processors that are responsible for the corners can start immediately. Conversely, the closer a tile is to the center of the domain, the longer the corresponding processor will remain idle. Additionally, those processors near the center of the domain are likely to encounter collisions, i.e. have multiple quadrants ready to be swept at the same time. For this situation we follow standard practice [8] and prioritize those directions that have the longest distance to sweep down the domain.

As discussed in Section 2.8, we wanted Soleil-X to be able to run fully on the GPU. The wavefront-like data dependencies of DOM do not lend themselves to an efficient GPU implementation, but we did manage to achieve performance on the same level as our best CPU code. Our implementation strategy consisted of decomposing each DOM sweep into a series of CUDA kernel launches, one per diagonal, starting from the appropriate source corner. Because the cells on a diagonal do not depend on each other they can all be processed in parallel. As CUDA kernels are executed in the order they are launched, we are guaranteed that every diagonal will find the values from the previous diagonal ready to be read. Additionally, we reorganized the storage order of the cell intensity values within a tile to enable memory accesses within CUDA kernels to be coalesced: the intensity values for the different angles within the same cell are stored contiguously, and the different cells are stored by diagonal. Finally, we were able to drastically reduce the memory requirements of the solver by recognizing that the intensity values on the faces do not need to be stored between iterations, and only the face values on the "frontier" of the sweep need to be tracked.

2.7 Multi-Domain Simulations

For some simulations we may want to couple two domains, with part of one serving as a boundary condition for the other. This connection involves two separate pieces:

- The values on the fluid cells of the first domain (velocity, temperature etc.) are directly copied into the matching cells of the second domain. Note that it is not desirable to simply override the fluid values on the second domain, as that would

most likely create an unphysical situation. Instead we define separate fields to hold these "incoming" values, and rely on the boundary condition handling code to incorporate them safely into the simulation.

- Particles in the first section are inserted into the appropriate sub-region of the second section, through the use of a third particles region, operating as a transfer queue. We partition this queue across the processors handling the first domain, with each processor getting a separate private part into which it copies its particles. Each tile on the second domain is then passed the entire queue, to read out of it those particles that it needs to import.

2.8 Solver Optimization

In this section we report on the methodology we followed for improving the performance of Soleil-X. We believe this approach is generally applicable to task-based implementations of scientific codes.

The factor that affected Soleil-X's time to solution the most has been the choices we made at the algorithm level. By simply choosing to use a coarser grid, narrower finite differencing scheme, or simpler radiation model we can gain more efficiency than most code-level improvements, at a fraction of the effort. However, only domain experts can decide whether this choice will adversely affect the quality of the results.

In the case of Soleil-X, the algorithmic change that produced the most improvement was to "stagger" the particle and radiation solve with respect to the fluid solve: In our simulations the fluid values change so fast that the particles cannot react immediately to the changes, so it is safe to consider the particles' reaction to the aggregate movements of the fluid around them over a period of time. Therefore, depending on the exact physics of the problem, we can get away with solving particles once every hundreds or even thousands of timesteps. Additionally, while particle positions remain constant, so do aggregate particle concentration values, meaning that the latest radiation solution can also be reused.

After a simulation scheme has been chosen, and the domain experts have decided

on the level of fidelity that is appropriate for the application, it is a good idea to first implement a clean, single-node version of the full simulation in a scripting language such as Matlab or Python. This sequential version of the code helps to gain an understanding of the data modeling requirements for each physics module, and what exactly is the interface between them. From this point we can start to design the transition of this code to a distributed environment, starting from the partitioning strategy. Once we have chosen how we want to split the data, most implementation choices become obvious. It is important to be mindful of how well the data modeling and partitioning choices for each module will work together, considering how the modules will be coupled in the full code.

After we have a distributed implementation in Legion, the most effective way to discover how to improve performance is to inspect the output of Legion's profiler for inefficiencies. The most common problems that we encountered and the most usual solutions were:

Task granularity too small Each task has little work to do, and thus completes very fast, so we cannot efficiently hide the overhead of the Legion runtime, or perhaps even the GPU kernel launch overhead. One solution is to fuse tasks where possible, to create larger units of computation.

High memory usage Each additional field we define on our regions requires additional memory to be allocated at runtime. We can reduce memory pressure by dropping redundant fields, e.g. those that can be re-computed on demand.

Extraneous instances We might observe that the runtime creates more physical instances of our regions than expected, most often duplicate instances for the same region. The most common culprit in this case is that the mapper is creating new instances instead of finding and reusing the existing ones.

Another common pitfall is the use of reduction privileges: When a task requests a reduction privilege on a region it is requesting the ability to reduce into *any* element of the region, regardless of the tile where that element normally resides. The only way for the runtime to support this functionality is to provide *each*

processor with a reduction buffer as big as the full region. In the common case where a processor will only reduce to elements within its assigned tile using read/write privileges will avoid the need for reduction instances.

Extraneous transfers We might observe that a dependency which we expected to be one-way actually causes data movement in both directions. This situation occurs most commonly because we chose to reuse the same region for multiple one-way transfers and did not inform the runtime that the previous contents of the region would be overwritten between transfers. The possible fixes are to do a fill operation to clear the contents of the region between transfers, or use write-discard privileges on the task that initializes the region.

Large transfers When we observe data transfers that are larger than expected, it is important to verify that our ghost regions are sized correctly.

Many small transfers If the data transfer timeline is full of small data transfers it may be beneficial to combine those into fewer, larger transfers, as that mode is more efficient for the hardware and Legion's DMA transfer sub-system. We can increase the granularity of transfers through the mapper, by artificially growing the amount of data we request from other nodes to include the extra data that we know will be needed later, or in the application code, by combining multiple tasks that perform different stencils on the same field into a single task.

Another, less common reason why there could be data transfers at small granularity is that we are doing explicit cross-region copies, but not mapping them such that we directly overwrite the data at the destination, in which case the runtime inserts a second transfer to do the actual overwrite when that data is next read.

In any case, it is important to allocate enough processors to the DMA engine, such that the processing of different transfers can occur in parallel.

Blocking in the main task We may notice a pattern where the work processors periodically stop executing tasks, followed by the main task resuming and runtime activity spiking.

The typical cause is that the main task is getting blocked and cannot run ahead to launch more work asynchronously. This might be because we have a branch in the main task that is dependent on a value derived from a task's output (even if the decision for that branch will not actually affect what tasks are launched), or we have inlined some output inside the main task, instead of doing it in a separate task.

It might also be the case that the main task is competing with other tasks for use of the same processor, in which case it may be worth simply allocating a separate processor to the main task.

High runtime activity If the processors devoted to the runtime analysis are highly utilized, the first thing to try is to allocate more processors to the runtime. Spawning the minimum number of runtime instances (one per node) will also serve to reduce the amount of analysis work, as there are fewer peers for each runtime instance to reason about. In the common case where the main loop of the simulation launches almost the same sequence of tasks every timestep, the tracing optimization [39] can be employed to amortize the cost of runtime analysis.

If the high runtime activity is specific to a single node (usually node 0), it is likely that this node is where all the control decisions are happening, and the node has become a control bottleneck. In that case the best approach is to rewrite the main task to take advantage of control replication [62], an optimization that distributes the main thread of control over multiple nodes.

Blocking due to data dependencies If processors are becoming idle until a transfer has completed, that is probably a sign that tasks on the critical path are not given priority. We could modify task priorities in the mapper such that the critical tasks execute as soon as they are ready, and other tasks can fill the processors while the required data transfers are occurring. If possible, it might also be beneficial to break up large stencils into independent, smaller stencils, to improve pipelining: The next stencil task can start executing while the transfers due to the previous stencil task are happening. Note that this change conflicts

with some of the advice given above; it is up to the programmer to judge what the best tradeoff is.

Another choice that we need to make is what computational resources to use for the simulation. Using both the CPUs and GPUs on a node is optimal, but comes at the cost of introducing communication between the corresponding memories, when the code running on one kind of processor needs to read data handled by the other. Therefore, it is important to split the workload considering not only the throughput of the different processors, but also the amount of communication necessitated by the split. For Soleil-X in particular, where the majority of the computation is throughput-bound, we run each simulation either entirely on the CPUs or entirely on the GPUs.

To have maximum flexibility in this decision it is useful to have both CPU and GPU versions of all tasks. This is true even for tasks where we expect the GPU implementation to be no better, or perhaps even worse, than the CPU one, since running everything on the GPU eliminates the need for data transfers between RAM and the framebuffer. Writing the application in Regent makes it much easier to support both processor kinds, because the Regent compiler can use a single task definition to generate both OpenMP and CUDA versions.

2.9 Solver Scalability

We were easily able to port Soleil-X to many leadership-class supercomputers (Titan [6], Piz Daint [3], Lassen [2], Sierra [4] and Summit [5]). Two features of the Legion platform were crucial for portability: the cross-platform nature of the runtime itself, and the Regent compiler's support for a wide range of processor and accelerator architectures through its use of the LLVM compiler library. Besides some basic adaptation of the mapping to each machine's node configuration, the actual tuning of the code was also transferable from one machine to the next.

In the remainder of this section we report on a weak scaling study we performed of Soleil-X, up to almost the full extent of the Sierra supercomputer. For this study we ran the full physics on GPUs. Each of the four Nvidia Tesla V100 GPUs of each

node was assigned 256^3 fluid cells and around 8M particles, enough to almost fill its framebuffer. We ran Soleil-X for 50 iterations on each configuration, solving full physics on each timestep (we disregard 5 warm-up and 5 ramp-down timesteps in our results). We chose to use the optically thin radiation model instead of DOM for this study, as the DOM method has inherent scalability issues that we did not want to conflate with the scalability of the main solver code.

Figure 2.10 summarizes the results of these runs. The 1-node case completed in 125 seconds. The solver demonstrated good weak scaling up to 256 nodes. What appears to be a drop in performance from 4 to 8 nodes can be attributed to an increase in required communication: When the fluid region is not split along some dimension (e.g. along the x dimension on a 1x4x4 tiling, which was used for the 4-node case, with a 1x2x2 block assigned to each node), there is no need to trade ghost cells on that dimension between iterations. Starting at 8 nodes (where we use a 2x4x4 tiling) every node has the maximum number of neighbors (two along each dimension, since we ran on a triply periodic domain), and thus performance stabilizes. The drop in performance after 512 nodes was traced back to scalability issues with Legion's implementation of Dynamic Control Replication; these issues and performance bugs should be rectified in the future.

2.10 Summary and Related Work

Overall, our experience developing Soleil-X on top of Legion and Regent was very positive. After an initial learning period, that involved getting used to the Legion asynchronous computation model and mapping interface, we were able to quickly ramp up to a working version of a solver, at a fraction of the time compared to past experience programming at the communication layer. We were then able to develop the DOM solver completely independently as a separate module and integrate it into the main simulation with only a small amount of additional code. We were able to port and scale the solver to many different machines and underlying architectures, with only minor changes to the application and mapper. Legion's profiler and the visualization tools we developed early were crucial in developing robust simulations.

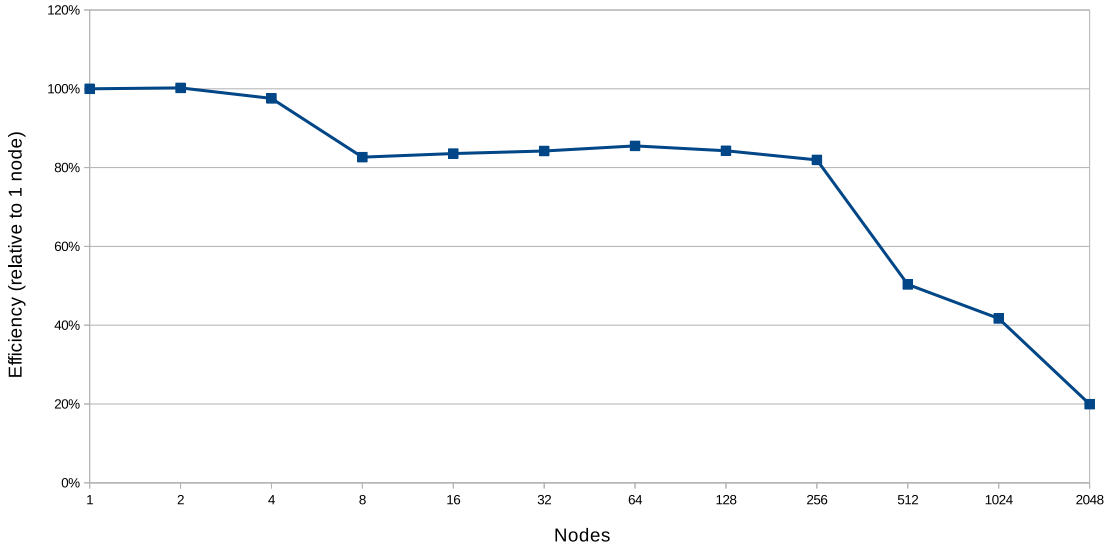

Figure 2.10: Results of weak scaling runs on Sierra

The Regent language provided a good tradeoff between being expressive and making all performance-relevant decisions accessible to the programmer. It thus served as a good common language between computer scientists and physicists; the physics experts could focus on the physical models, while the computer scientists worked on optimizing the code. The conciseness and sequential appearance of the source code, coupled with the parallelizer's transparent handling of ghost regions, allowed for easy debugging. Finally, the Regent compiler's CUDA code generator made it easy to add GPU support.

Regarding work that remains to be done, the feature that would expand the capabilities of Soleil-X the most is support for implicit integration methods, which will require an efficient Legion implementation of a linear equation solver. At this time we are focusing on extending Soleil-X with support for chemistry, particle collisions and more complex boundary conditions, and optimizing existing components, particularly the wavefront computation pattern of the DOM solver. On the Legion side, efforts to make simulation development easier should focus on providing more visibility into the decisions of the runtime (and explaining what prompted these decisions, and how they affect performance) and making the Regent compiler more robust and

efficient. Implementing domain coupling could be made significantly simpler with library support and certain improvements to the runtime, such as faster repartitioning and support for sparse physical instances.

A variety of multi-physics solver frameworks are currently available [64, 70, 33, 10, 50, 27, 35], each with its own feature set and parallelization capabilities. Soleil-X is, to our knowledge, the first such system built with a task-based runtime in mind.

The closest system to ours is Uintah, which has been extended from its original design to support asynchronous DAG-based execution [44]. Uintah targets simulations that involve fluid, particles and chemistry, It operates on a structured mesh and uses implicit integration methods. The full mesh is divided into hexahedral *patches* which are distributed to processors, and can be refined independently. Uintah's tasking system is customized for simulation programs, exposing two versions of each field (for the state before and after each timestep) and restricting cross-patch communication to the trading of ghost cells.

Techniques for coupling different physical domains in simulations have been studied since the dawn of scientific computing, e.g. see the Particle-In-Cell [23] and Multiphase Particle-In-Cell [11] methods.

Chapter 3

Uncertainty Quantification

In this chapter we discuss the potential of task-based runtimes to improve the practice of Uncertainty Quantification. In particular, we will exploit the compositionality of the Legion task-based system to build a UQ framework directly on top of the Soleil-X solver.

3.1 Background

For the purposes of this chapter assume that we have some physical system of interest and want to computationally predict the value of a single metric (Quantity of Interest, QoI) related to the behavior of that system. Any numerical simulation we might use for this purpose is, at a fundamental level, just a deterministic computer program implementing a mathematical function f, that accepts a set of inputs ξ (specified at machine precision) describing the initial state of the system and outputs a prediction $Q = f(\xi)$ for the QoI. With the appropriate choice of sophisticated modeling techniques, this prediction can be made arbitrarily accurate. However, this increased accuracy does not necessarily translate to predictive power over the original physical system, for the simple reason that, in practice, most of the inputs to the system cannot be measured with perfect precision, and their values may even change every time the system is observed. Methods from the field of Uncertainty Quantification (UQ) tackle this problem by modeling the uncertain inputs as random variables drawn from

a probability distribution, and quantifying their impact on the QoI.

Monte Carlo [30] (MC)-based random sampling methods are a popular category of UQ methods because they are broadly applicable to a variety of simulations, although they are mostly restricted to obtaining descriptive statistics of the real system's QoI. Let $\mathbb{E}[Q]$, $\mathbb{V}ar(Q)$ and σ_Q denote the mean, variance and standard deviation of the QoI. We can use MC sampling to derive an estimate of $\mathbb{E}[Q]$ as follows: Draw a sample of N independent values, denoted $\xi^{(i)}$ where $i \in \{1, \ldots, N\}$, from the input probability distribution. Then the MC estimator for $\mathbb{E}[Q]$ is defined as $\hat{Q} = N^{-1} \sum_{i=1}^{N} f\left(\xi^{(i)}\right)$. Although unbiased, the accuracy of this estimator, measured by its standard deviation $\sigma_{\hat{Q}} = \sqrt{\mathbb{V}ar(Q)/N}$, decays slowly as a function of N. Therefore, reaching an acceptable level of accuracy may require a prohibitively large number of evaluations of the (potentially quite expensive) base simulation.

It is possible to improve on this tradeoff, by exploiting the fact that in computational science multiple numerical models of the same system can be constructed, at varying levels of accuracy and evaluation cost. For our purposes we consider two "tiers" of models: computationally expensive High-Fidelity (HF) models that accurately represent the underlying physical phenomena, and relatively cheaper Low-Fidelity (LF) models that are possibly less accurate by themselves, but their response to input variability is typically well-correlated with the HF. Various MC acceleration techniques [52, 24] then attempt to combine the accuracy of HF models with the speedup of LF models to obtain a more accurate statistical estimator than would be possible with pure MC sampling of the HF.

We will focus our attention on a simple (but popular) MC acceleration technique, the method of Control Variates (CV) [51, 48], also known as Multi-Fidelity (MF). This approach replaces the function f that describes the HF with $f + \alpha(g - \mathbb{E}[g])$, where g is the function that describes the LF, and is expected to be highly correlated with f. To construct the CV estimator we again draw a sample $\xi^{(i)}$ of N input values and use it to construct MC estimators for both the HF and LF (denoted $\hat{Q}^{\mathrm{HF,MC}}$ and $\hat{Q}^{\mathrm{LF,MC}}$ respectively). The CV estimator is then defined as:

$$\hat{Q}^{\mathrm{CV}} = \hat{Q}^{\mathrm{HF,MC}} + \alpha\left(\hat{Q}^{\mathrm{LF,MC}} - \mathbb{E}\left[Q^{\mathrm{LF}}\right]\right) \tag{3.1}$$

Here, the population mean $\mathbb{E}\left[Q^{\mathrm{LF}}\right]$ of the LF model g over the full input distribution is not generally known. Instead we need to approximate it by means of another MC estimator, which evaluates the LF on an additional rN inputs to compute:

$$\mathbb{E}\left[Q^{\mathrm{LF}}\right] \approx \frac{1}{N(1+r)} \sum_{i=1}^{N(1+r)} g\left(\xi^{(i)}\right) \tag{3.2}$$

The population variances $\mathbb{V}ar\left(Q^{\mathrm{HF}}\right)$ and $\mathbb{V}ar\left(Q^{\mathrm{LF}}\right)$ and covariance $\mathrm{Cov}\left(Q^{\mathrm{HF}}, Q^{\mathrm{LF}}\right)$ are simply approximated using the corresponding sample metrics from the N original MC runs. The parameter $\alpha = -\mathrm{Cov}\left(Q^{\mathrm{HF}}, Q^{\mathrm{LF}}\right) / \mathbb{V}ar\left(Q^{\mathrm{LF}}\right)$ is chosen to minimize the variance of $\hat{Q}^{\mathrm{CV}}$, by requiring $\mathrm{d}\mathbb{V}ar\left(\hat{Q}^{\mathrm{CV}}\right)/\mathrm{d}\alpha = 0$. This optimal α selection leads to:

$$\mathbb{V}ar\left(\hat{Q}^{\mathrm{CV}}\right) = \mathbb{V}ar\left(\hat{Q}^{\mathrm{HF,MC}}\right)\left(1 - \frac{r}{1+r}\rho^2\right) \tag{3.3}$$

where $\rho = \mathrm{Cov}\left(Q^{\mathrm{HF}}, Q^{\mathrm{LF}}\right) / \sqrt{\mathbb{V}ar\left(Q^{\mathrm{HF}}\right)\mathbb{V}ar\left(Q^{\mathrm{LF}}\right)}$ is the Pearson correlation coefficient between the HF and LF models.

3.2 UQ and Task-Based Programming

At their core, UQ frameworks need to support the execution of large ensembles (groups) of samples (copies of the simulation at different levels of fidelity, and for different choices of uncertain inputs). This type of application, due to its high level of parallelism and diversity of computational requirements, is ideal for task-based systems. We believe that task-based systems have the potential to significantly improve the time required to perform UQ studies, not just by improving how efficiently ensembles can be executed, but also by automating many aspects of UQ that are currently performed by hand, and possibly even enable new UQ algorithms. The potential benefits of task-based systems for UQ include:

Flexibility A task-based runtime, and the Legion runtime in particular, can scale to ensembles of any size, can accommodate arbitrary combinations of samples at different levels of computational complexity, and allows easy porting to different

machines.

Shared context Having all the samples execute under a single runtime exposes more opportunities for parallelism than running them as separate jobs. We can achieve better utilization on each node by interleaving multiple samples, and across the ensemble by using the smaller samples to fill idle time. Depending on the simulation, it might even be possible to share redundant computation between compatible samples.

Improved scheduling With a sufficiently sophisticated mapper we can maximize the utilization of the machine, by allocating samples to machines according to their computational requirements and scaling behavior (see Chapter 5 for further discussion). Additionally, we could utilize all the processors on a node (both CPUs and GPUs) without incurring communication costs, simply by placing separate samples on each.

Dynamic information Legion's profiling capabilities allow us to collect accurate runtime information, to be used as cost estimates by the UQ algorithms. It is also easy to augment task-based simulations with arbitrary monitoring code (e.g. point probes and whole-domain statistics output), as the runtime will automatically take care of the required data transfers and scheduling.

Dynamic mapping A fully dynamic mapper could detect and handle load imbalance on the fly, by migrating samples and modifying node allocations as needed. It could also handle a dynamically changing set of samples, which would open the way for new, dynamic UQ algorithms.

Integrated analysis and execution In current practice the statistical analysis of UQ results is typically done manually and separately from ensemble execution. Using Legion we can integrate this analysis with the running of the ensemble, which would allow us to make automatic, early decisions about what samples to include and exclude from an ensemble, including stopping samples on the fly, and creating new ones according to the needs of the algorithm.

Fault tolerance Using Legion's resilience infrastructure we can recover gracefully from the failure of a few samples, and even adapt their parameters on the fly (e.g. the timestep size or size of the grid) if we detect that they failed because of issues with the physics resolution.

For this project we used Soleil-X to experiment with the design of an ensemble computation framework over a task-based system. To achieve basic UQ support in Soleil-X we simply added another level above the main task of the simulation, that can launch multiple copies of it and run them under a single runtime instance; this resulted in an additional level of parallelism being added to the application (see Figure 3.1). Our UQ framework supports any number of samples of different fidelities executing independently (but potentially on the same set of nodes) using both CPUs and GPUs. The parameters of the different samples are currently read from external configuration files. Multiple samples execute concurrently under a single runtime instance, thus allowing Legion to better utilize the available hardware. We support the dynamic collection of timing information, volume averages and point probes. Our mapping strategy is currently static and user-controlled. Exploring the rest of the above listed features is ongoing work.

3.3 Related Work

UQ studies tend to make use of one-off implementations of their chosen method, but a number of generic UQ libraries do exist; we list some of the major ones below. To our knowledge, none of the currently available UQ libraries can exploit parallelism beyond the level of running different samples concurrently, because they have only black-box access to the simulation being studied.

Dakota [7] is a C++ library that implements a variety of statistical functions, optimization algorithms and sampling methods, that can be combined to build complex UQ algorithms. Dakota only controls the outer loop of the UQ algorithm, and invokes the simulation under study as a black box. It includes some support for parallel execution of different samples through an MPI interface.

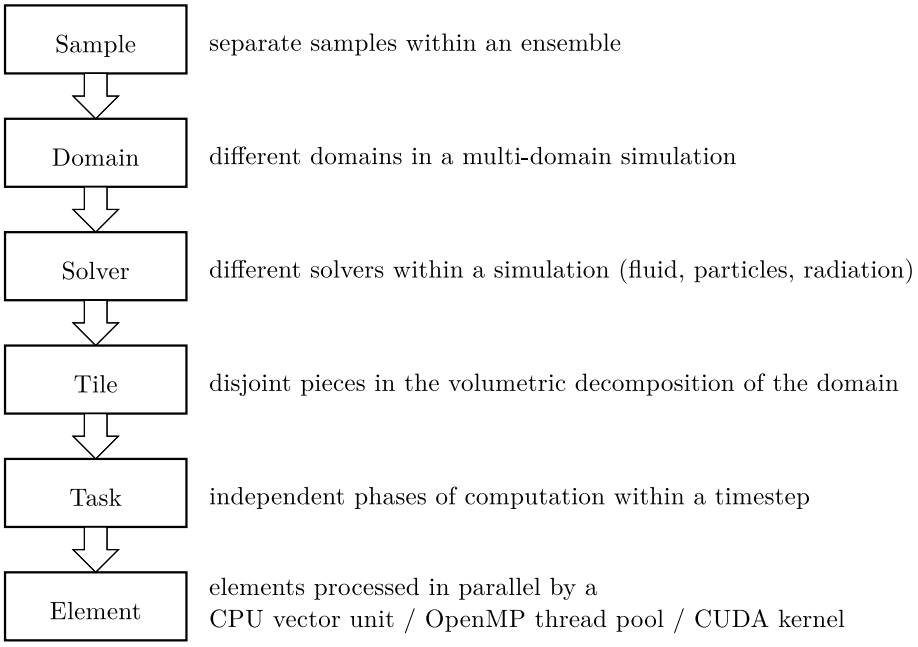

Figure 3.1: Levels of parallelism within Soleil-X with UQ support

QUESO [56] implements a small set of UQ methods that the user can apply to their simulation by defining certain callbacks. QUESO can use multiple processors on a single node, or multiple nodes over MPI, to distribute the simulation or the statistical analysis.

UQLab [42] is a Matlab-based UQ framework aimed at non-highly-IT trained scientists. Its design focuses on modularity and ease of use over performance. It includes some simple facilities for invoking the simulation over distributed machines.

Of the existing UQ libraries the one most related to our work is Π4U [28], which builds on top of an MPI-based tasking library called TORC [29] to implement various sampling and optimization algorithms. TORC exposes an RPC-like tasking interface with explicit handling of dependencies, and can utilize both multi-core CPUs and GPUs. The framework transparently handles scheduling, load balancing, work stealing, task and data movement. As with the other libraries listed above, Π4U treats the simulation being studied as a black-box, and thus cannot exploit any parallelism within the simulation code itself.

Chapter 4

Optimal Low-Fidelity Model Search

4.1 Introduction

One of our goals in this project has been to explore the potential of task-based programming to improve the productivity of UQ practitioners. One way to do this is to exploit the capabilities of task-based runtimes to automate more of the tasks that UQ practitioners currently do by hand. One major such task is actually picking which LF model(s) to include as part of a UQ study.

The standard practice currently is to leave it up to the UQ expert to select a combination of model coarsening options that, based on the expert's experience and intuition, are expected to provide a good balance of speedup and fidelity. In the case of multi-physics problems this choice is hard to get right, for two reasons: First, most of the physical phenomena that such solvers simulate exhibit strongly non-linear behavior, which, coupled with the the unpredictable interactions between the different physics modules, makes it almost impossible to predict how the overall LF will behave compared to the HF. Second, while the performance improvement associated with a coarsening choice seems straightforward to predict (e.g. reducing the size of the grid by half should approximately halve the running time of the fluid solve), these effects are inherently tied to a specific machine and set of implementation choices. For example,

we may perform the same UQ study on a different machine with more powerful GPUs, which favor the fluid solver but not the radiation solver, meaning that coarsening the fluid mesh will now have less of an effect on the overall performance of the code. Therefore, the user's experience on one machine will not necessarily translate to a different machine, or even different version of the code, and might be more misleading than helpful.

Based on the observation that user intuition is a poor predictor of the suitability of an LF model, we instead propose that computational power be brought to bear as a better alternative. We will show that, using some computation time strategically, and a little input from the user, we have the potential to make better modeling choices than human experts.

Specifically, we propose a generic, search-based approach for selecting the optimal LF model for any combination of UQ algorithm and HF simulation. Our method treats the base simulation code as a black box; all we require is an interface to launch the simulation using a selected LF model and collect the QoI from that execution. We also expect from the user to define a sampling of uncertain inputs to evaluate and the dimensions on which we can coarsen the simulation, with appropriate ranges for each (the available coarsening parameters are specific to a simulation, and therefore some creativity is involved in exposing them as tunable knobs; the more ways we have to coarsen the simulation, the more likely we will be able to find a good LF model). Then we can simply launch an automated search over the space of coarsening options, to find good LF models.

Judging the optimality of an LF model is not as straightforward as it may seem, because there is typically more than one metric for comparing different LFs, depending on the UQ method. Take, for example, the CV method. Based on Equation 3.3, the higher the correlation ρ of the LF with the HF the higher the achieved variance reduction. Conversely, the higher the cost of the LF compared to the HF the lower the ratio r of additional LF runs we can afford to do, and the lower the achieved variance reduction. In total, the correlation and computational cost of each LF model both affect its usefulness, and there is usually a tradeoff between these two values, meaning that two different LF models are very likely incomparable. What is then

interesting to find is which LFs represent the best tradeoff between cost and correlation (the so-called "Pareto-efficient" ones): For any desired level of correlation there are possibly multiple choices of LF, and we would like to find the most efficient one. If we know the exact parameters of the UQ study to be performed (e.g. the assigned computational budget), we could potentially work through the UQ method's formulas to combine all of an LF model's metrics into a single number: expected variance reduction.

In the following sections we present a study that we performed as a proof of concept for this approach. We take a specific choice of UQ method (CV) and HF simulation (implemented on Soleil-X and augmented with a set of configurable coarsening parameters) and perform a simple exhaustive search to identify the Pareto-efficient choices for LF models. As analyzed above, given the choice of CV as the UQ method, LF models are compared in terms of computational cost and correlation with the HF. Based on the results of this search we show that (a) the choice of LF model may not be trivial, and (b) an automated method can outperform human experts in LF model selection. Given that we achieve these results with such a simple premise and choice of search strategy, we are confident that this method has significant potential to improve the state of the art in the practice of performing UQ studies. In Section 4.7 we discuss ways to improve the efficiency of the basic search methodology. Future work on this topic could explore more sophisticated search schemes, e.g. combining the time devoted to LF model searching and the actual running of the UQ ensemble into a single computational budget. We expect that such extensions will be able to take even better advantage of the power and flexibility of task-based runtimes than our initial study, which simply relied on Legion for the scheduling and efficient execution of multiple simulation instances.

4.2 Setup of LF Search Study

For this study we repurposed a simulation setup from previous work [57], making only minor changes to fit the capabilities of our solver. We chose this particular case because, while it has relatively low computational requirements, it involves a variety

of potentially uncertain inputs that all meaningfully affect the result of the simulation. It also makes full use of the capabilities of our solver, and thus gives us a lot of options for reducing the fidelity of the simulation. Finally, it is a well-characterized problem, of potential interest to other scientists.

The problem under study involves two domains: The "HIT" domain is a triply-periodic cube with side $W = 0.04m$ seeded with around 2M monodisperse (identical) particles and HIT-forced to generate particle-laden turbulent flow. A single-cell-wide x-slice from this domain is periodically sampled and used as input to the "channel" domain, which is a $L \times W \times W$ "duct" with $L = 0.16m$, periodic on the y and z dimensions, and using inflow-outflow boundary conditions on x. The fluid-particle mixture flows through this domain at a rate of $U_0 \approx 2m/s$ and gets radiated in the y direction from both sides. The length of time L/U_0 that fluid takes to flow down the full length of the duct is called a Flow-Through Time (FTT). X-slices are copied once every dx/U_0, where dx is the length of a single cell, to give the previously inserted particles enough time to clear the inlet.

We use the same QoI as the original paper: the heat transfer coefficient, defined as the time-averaged temperature difference between fluid and particles at the outlet plane (relative to the starting temperature). To record the value of this QoI we first run the simulation for 1 FTT, enough to allow the turbulent mixture copied from the HIT domain to reach the outlet. We then start recording for every iteration the average fluid and particle temperature over the final (outlet) slice of the channel domain, and use those averages to compute the value of the QoI for that iteration. The final QoI is calculated by averaging the per-iteration QoI values over the rest of the simulation. This final time-averaging step is necessary because, by virtue of the turbulent nature of the flow, the QoI is not actually expected to reach a steady state, but instead oscillate around a value, which the time average should converge to.

We consider six of the simulation's parameters as uncertain inputs: Reynolds number (how much turbulence the fluid tends to develop), Stokes number (how responsive a particle is to the fluid movement around it), mass loading ratio (how rich the fluid-particle mixture is), radiation intensity, heat capacity of the particles, and heat transfer ratio from particles to fluid. Each of these values is assumed to be

uniformly distributed within a ±5% range around a nominal value (in an actual UQ study the range and probability distribution of each uncertain input would be determined separately, based on physics calculations or experimental data). Using a Kernel Density Estimation-based sampling algorithm [58] we selected 32 combinations of these parameters, a sample size large enough for the aggregate statistics of the QoI to have sufficiently converged (see Figure 4.1).

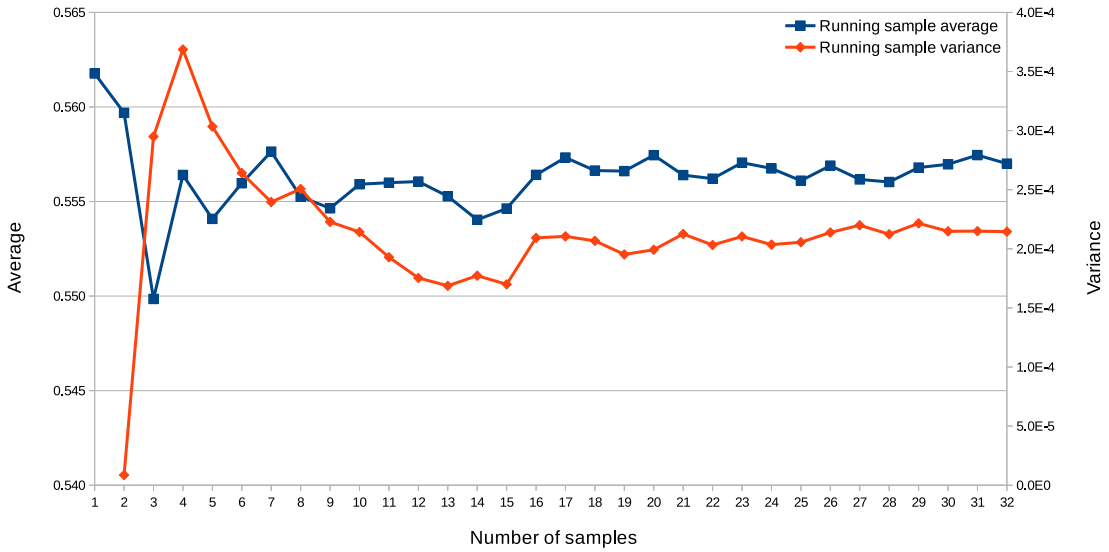

Figure 4.1: Evolution of HF sample statistics as sample size grows

We explore 11 different dimensions for reducing the fidelity of the base simulation: coarsening of the fluid or radiation grid on each of the x, y and z dimensions, reduction in the number of solid angles used in DOM, use of the optically-thin radiation model over DOM, use of a lower-order Runge-Kutta model, reduction in integration time, and size of particle parcels. The last parameter refers to the technique of surrogate particles [65], whereby the number of particles being simulated is reduced by grouping them into particle parcels, each of whom represents a group of particles from the original simulation. Particle parcels follow the same equations as normal particles, with some factors weighted appropriately for the size of the parcel.

We restrict our search space to only two choices per coarsening parameter, as listed in Table 4.1. Even with this small amount of choices, our search space contains

a total of 1088 different LF models.

Parameter	HF setting	LF choices
Flow grid: #cells in x	512	128, 64
Flow grid: #cells in y	128	32, 16
Flow grid: #cells in z	128	32, 16
Particle parcel size	1	10, 100
Radiation model	DOM	DOM, opt. thin
DOM grid: #cells in x	256	64, 32
DOM grid: #cells in y	64	16, 8
DOM grid: #cells in z	64	16, 8
DOM: # solid angles	350	86, 50
Runge-Kutta order	4th	4th, 3rd
Number of FTTs	3	3, 2

Table 4.1: LF model search space. Grid sizes refer to the channel domain.

The actual search process starts by running the HF configuration on our set of 32 samples and recording the output QoI and computational cost of every sample. Each of the candidate LF models is then evaluated on the same set of samples and its correlation with the HF estimated using the sample correlation and covariance formulas. We use the actual wall-clock running time of each model as its computational cost (averaged over its 32 evaluations over all samples).

4.3 Validation of Base Simulation

Before we could use the proposed simulation as a case study for our LF model search method, we first needed to verify that it behaves correctly under a variety of inputs and coarsening configurations. Fully validating a multi-physics simulation such as Soleil-X, however, is impossible in practice: The equations involved are too complex to solve analytically, so there is no ground truth to compare with. Additionally, the simulated phenomena are chaotic, meaning a small difference in the inputs (on the order of machine floating point precision) can cause a large divergence in the outputs. Therefore, the process of building confidence in the solver will necessarily be imprecise. In this section we outline the methodology we followed for validating

our case, given these constraints.

The first step in gauging the behavior of a simulation is to visualize a time progression and have a domain expert inspect it visually. Although subjective, this process is a good sanity check for verifying that all the pieces of the simulation are working together correctly. Some of the features the expert would look out for is that turbulence develops in an expected pattern, particles form loose clusters, and the movement of particles loosely follows that of the fluid.

As a more automated check of the simulation, we can take averages of simulated values over appropriate subsets of the domain (e.g. the channel inlet and outlet, or an x-slice crossing the domain) across time, and verify that their trends are within the bounds suggested by the underlying physics. Some of these checks include:

- The average velocity across the entire channel domain remains constant at U_0.
- The average temperature of the fluid over an x-slice increases roughly monotonically from inlet to outlet.
- After some time the average temperature at the outlet has stabilized (we should see it oscillating around a value, rather than continuing to increase).
- The average kinetic energy and dissipation in the HIT domain eventually converge to their target values, as set in the configuration of the HIT forcing component.

At the level of the full UQ run we additionally check the following:

- The output QoI (heat transfer coefficient) is, on average, close to the value computed in the original paper.
- The simulation responds to changes in the input uncertainties (the output QoI exhibits a variability of 5-10% across the range of inputs).
- Our search space contains at least some LF models that have high enough correlation with the HF that they would be useful for UQ studies (at least 0.8).

- There exists a good mix of LF models within our search space, in terms of both computational cost and correlation, and the relationship between these two metrics is not trivial.
- Running the simulation twice on the same configuration should give almost identical results in terms of the QoI. This suggest that the uncertainty in the value of the QoI is purely due to input variability and not randomness in the initial conditions. Note that we only require stability in a statistical sense; we do not expect two runs of the simulation to agree timestep-by-timestep.

4.4 Issues Encountered

As part of performing our study we had to run our solver in a wide variety of configurations and input conditions. This process stress tested the solver in unexpected ways and uncovered a surprising number of issues with our base solver, our case setup, and even the Legion runtime and Regent compiler. Working through these issues to make our setup robust enough to handle the full scale of our search was by far the most time-consuming part of this study. In this section we report on the major issues we encountered while setting up our study, and the lessons learned from each.

The issues we encountered tended to fall into one of the following categories, in order from most to least frequent:

- caused the solver to reach an extreme physical situation and diverge; this was particularly common with the code that handles the boundary conditions at the outlet, as the physics behind that code is very sensitive to extreme values
- software bugs in Soleil-X (such as missing a particle cell pointer update, or not restoring the full simulation state after a restart)
- bugs in the Regent compiler (mainly the parallelizer)
- bugs in the Legion runtime
- transient issues due to misconfigured nodes

The first major issue we discovered with our simulation setup was immediately obvious upon inspecting a visualization of the particle positions in the channel domain: The particles were forming "streaks", whereupon every particle was followed by another particle at almost exactly the same y and z coordinates, and only a small distance behind it in x. The cause of this issue was traced back to the way we were coupling the two domains: The original version of our code was using the HIT forcing technique to develop turbulence in the HIT domain with an average velocity of zero, and the copied slice was artificially augmented with an additional velocity of U_0 before getting copied to the channel domain. Because the velocity fluctuations due to turbulence were much smaller than U_0, by the time dx/U_0 seconds had passed and we had to copy a slice from the HIT domain into the channel domain, the positions of the particles in that slice had hardly changed, so we would end up with "freeze-frame" particles going down the channel. The solution we came up with was to pre-impose an average velocity of U_0 in the HIT domain (over which we would add turbulence), so that the two domains would be synchronized. Simply initializing the HIT domain with a uniform velocity of U_0 did not fix the problem, as the HIT forcing algorithm would naturally tend to bring the average velocity down to zero, as it tried to reach its target level of kinetic energy. Instead we worked around the limitations of the HIT forcing method by subtracting U_0 from every cell's velocity before applying the forcing term, then adding it back afterwards.

Following our fix to the previous issue, we realized that the average velocity in the channel section was no longer constant at U_0. Instead we were seeing the velocity increase or decrease over the course of the simulation, depending on the (random) initial conditions. The reason was that the velocity in the HIT domain was free to evolve according to the equations of motion, and would thus sometimes develop a tendency to increase or decrease on average. The solution was to force the velocity in the HIT domain to remain at a constant average of U_0. At the end of each timestep we would compute the average velocity across the domain, and calculate its difference from the target average of U_0. We would then apply to every cell and particle in the domain a small correction to counteract this divergence, before proceeding to the next timestep.

Another issue we discovered by inspecting a visualization of the channel domain was that particles seemed to be clustered on approximately the top half of the channel, with the bottom half empty. This behavior did not occur with the HF, only the LFs. After a little investigation we traced the issue back to the code that initialized the particles in the HIT domain: We were allocating particles sequentially to cells, filling an entire x-slice before moving to y, then filling an entire xy-plane before moving to z. In the case of the HF there were about as many particles as cells in the HIT domain, which meant the initial distribution was approximately uniform. However, the LFs were making use of particle parcels, which meant there were 10 to 100 times fewer simulated particles, therefore the particle initialization code ended up filling only the top half of the HIT domain, and this distribution would get copied over to the channel domain. To solve this we simply switched to randomized particle initialization, which produces an approximately uniform concentration of particles across the HIT domain, regardless of the number of particles. This issue demonstrates an important requirement of coarsening methods: applying the coarsening should produce a problem that is analogous to the original problem (but scaled down), otherwise the solution to this new problem is unlikely to be correlated with the solution of the original.

The following issue took us the longest to fix, as we were hitting upon the limitations of the outlet boundary condition handling code, which we could not fix without diving deep into the physics behind it. During some of our runs the fluid flowing in from the channel section would randomly develop a large, high-velocity eddy at its front, probably due to concentrated absorption of radiation, as a result of particle clustering. This behavior is not unexpected, and it is not surprising that it occurs non-deterministically; certain combinations of uncertain inputs favor the formation of such eddies more than others. When this eddy would reach the outlet, its large velocity and temperature values would cause sharp gradients against the previous values at the boundaries, causing the boundary condition handling code to diverge. Since there was little we could do to improve the stability of this code, we had to work around its limitations: Instead of initializing the channel domain to be empty, we initialized it with particles at the same concentration as the HIT domain, all following

the fluid as it flows down the duct at U_0. With this change the initial conditions of the fluid-particle mix in the channel became closer to the incoming mix, and thus the gradients became smoother. Our solution to this problem demonstrates a more general principle, that we have some flexibility in how we set up our case, without meaningfully changing the problem we are solving (in our case the original contents of the channel domain would get cleared after 1 FTT anyway, and we only care about the behavior of the system after that point).

Finally, we note that it is important to not set the value of a coarsening parameter so low that the physics of the problem are under-resolved, otherwise instability or unphysical results might occur. These minimums can usually be derived by applying certain formulas to the initial conditions of the problem, that capture the scales of the phenomena which are expected to develop. We list here some of the more prominent physics-related minimums we uncovered while setting up this case.

- The timestep we use cannot be too large, otherwise the fluid solver will diverge. Soleil-X can adjust its timestep dynamically as the values of the simulation change, through a CFL condition check. However, this check is not always precise, and its computation causes blocking in the main task, so we generally try to avoid it if the temperature in the domain (the main parameter that controls maximum step size) is not expected to vary significantly. Instead we use the maximum temperature in the domain to pre-compute a safe setting for the timestep. It is important, however, to account for the maximum temperature that we expect to occur anywhere in the domain (even due to local fluctuations), over the entire duration of the simulation.

- The grid on the channel domain cannot be too coarse in the x direction (no fewer than $\sim$32 cells), otherwise the equations managing the inlet and outlet planes become coupled, and backwards flow starts to develop.

- The grid on the channel domain cannot be too coarse in the y or z directions (no fewer than $\sim$16 cells in either direction), so that every large eddy that might develop will have at least 3 cells across it. If a large eddy can fit inside two cells

then we will have neighboring cells at opposites sides of the eddy fluctuating in opposite directions, producing very large gradients.

- Setting the particle parcel size too high can cause instability, since every simulated particle now behaves as multiple particles in a tight clump that collectively absorb a lot of radiation and heat up their containing cell, thus causing high temperature gradients with that cell's neighbors.

4.5 Results of LF Search

The results of our LF search study are summarized in Figure 4.2. Each LF we considered is plotted in terms of correlation with the HF and execution time. For reference, the running time of the HF was 526,208s. The search discovers 4 LFs on the Pareto front, each at a correlation of over 0.92, and at least 72 times faster than the HF. The parameters and exact scores of these LFs are given in Table 4.2. We performed all of our runs on the Lassen supercomputer, running Soleil-X in full-GPU mode and allocating one Nvidia Tesla V100 GPU to each sample.

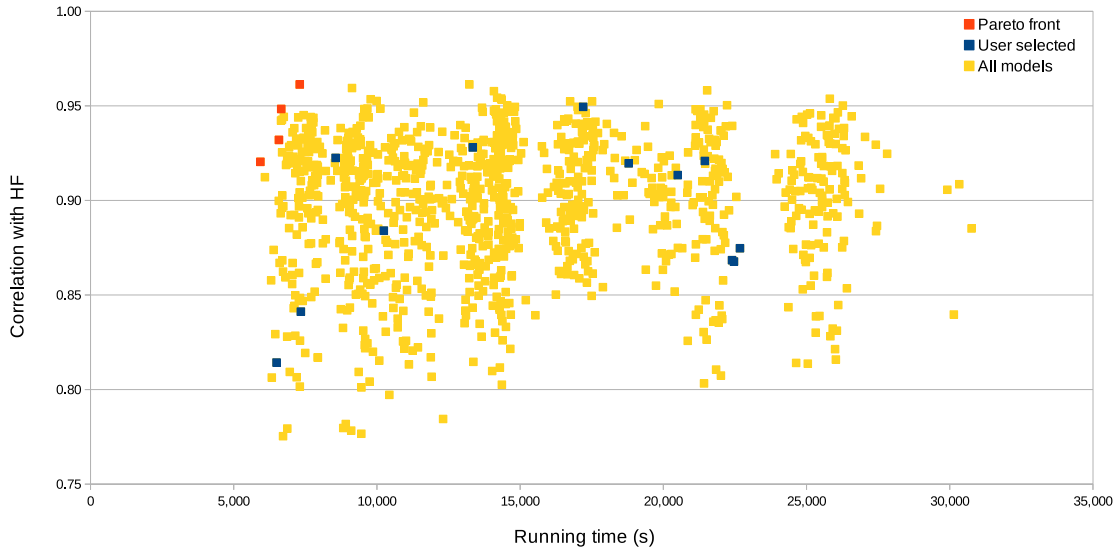

Figure 4.2: Results of LF search study. Each point represents a different LF within our designated search space.

Parameter	LF1	LF2	LF3	LF4
Correlation with HF	0.92043	0.93200	0.94837	0.96131
Running time (s)	5,930	6,579	6,658	7,306
Flow grid: #cells in x	64	64	64	64
Flow grid: #cells in y	16	16	16	32
Flow grid: #cells in z	32	32	32	16
Particle parcel size	10	10	10	10
Radiation model	opt. thin	DOM	DOM	DOM
DOM grid: #cells in x	–	32	32	32
DOM grid: #cells in y	–	8	8	8
DOM grid: #cells in z	–	8	16	8
DOM: # solid angles	–	50	50	50
Runge-Kutta order	3	3	3	3
Number of FTTs	2	2	2	2

Table 4.2: Pareto front of LF model search space

To get an idea of the efficiency of the LFs on the Pareto front consider that performing one extra run of the HF on a different uncertain input (over the original 32) would only reduce the MC estimator's variance by 3.03%. In the same amount of time we could instead perform 88, 80, 79 or 72 runs respectively of one of the LFs. Combining the results of these LF runs with the original 32 runs of the HF (following the CV method) we would achieve a 54.17%, 52.11%, 53.52% or 51.36% reduction in estimator variance respectively. We specifically executed LF4 a total of 72 times (the 32 required to compute correlation with the HF, plus an additional 40), to go from an original HF-only MC estimate of 0.55700 with a standard deviation of 0.0025891 to a combined CV estimate of 0.55757 with a standard deviation of 0.0018061.

Note that the optimal choice of LF for a CV execution depends on the computational budget available for running the LFs, as can be seen in Figure 4.3, where we plot the expected variance reduction of the four Pareto-efficient LFs for different budgets.

Compared to our experts' experience doing actual UQ studies, most of the LFs in our search space were surprisingly well-correlated with the HF. Therefore, while our results point to the possibility that systematic search is simply able to find more highly correlated LFs than human expertise, it is also possible that our study is not

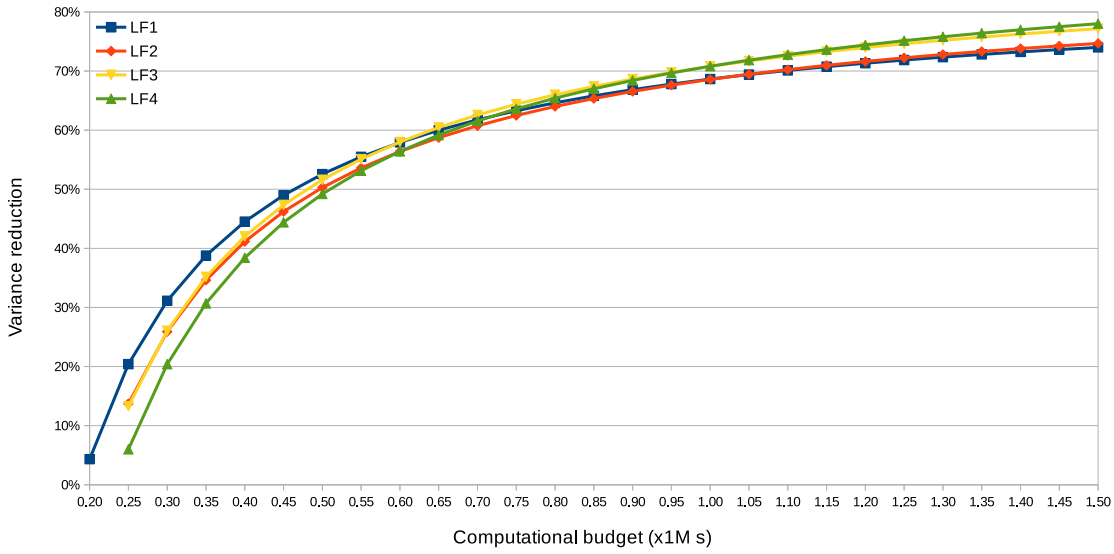

Figure 4.3: Relative efficiency of Pareto-efficient LFs for different budgets, in terms of achieved estimator variance reduction when used together with 32 evaluations of the HF in a CV execution. Note that on the lowest budget setting the three more expensive LFs cannot even be executed the requisite 32 times to compute their correlation with the HF.

fully representative of every situation a user might encounter in practice, in particular simulations and QoIs that do not respond as well to coarsening.

The LF models we explored also exhibited subpar performance gains compared to usual experience, achieving only a ∼90x speedup, where a more common figure is 100x – 1000x. The reason for this can be traced back to the same reason that LFs did not vary significantly in performance: Owing to the high performance of the hardware (GPUs) we were running on, even at the HF level our workload was close to the minimum size where the amount of useful work could hide the overhead of the communication, runtime analysis and GPU kernel launching. We note that this fact does not in any way invalidate the usefulness of our approach; in fact it serves to remind us that, as already mentioned in Section 4.1, the performance gains of a coarsening choice are inherently tied to a specific machine, implementation, and mapping policy. Therefore, a user's intuition regarding the expected benefit of any such choice does not necessarily translate from one setting to the next, while an automated method like ours can be easily re-run in a different setting to produce an accurate estimate.

4.6 User Study

In this section we present a user study that demonstrates our method's potential to improve current UQ practice.

We provided a number of scientists from Stanford's Mechanical Engineering Department with a description of our problem and the coarsening parameters explored by our LF model search, and asked them to pick a LF model to use in a hypothetical UQ study. The users we selected were a mix of graduate students, postdoctoral scholars and faculty, not necessarily experts in UQ, but rather physicists that might need to use UQ in the course of their work (and thus need to make an informed decision on what LF model to use).

The choices of these users are summarized in Figure 4.2. We note that (a) every person made a different choice, (b) the users' choices varied significantly, in both correlation and performance, and (c) none of the users selected a LF particularly

close to the Pareto front. The average user-selected LF has a correlation of 0.89202 with the HF, and is only 33x faster. Using such an LF in a CV evaluation in place of a 33rd HF evaluation would result in a variance reduction of 2.36%, worse than simply running the additional HF.

While our study obviously covers only a single simulation and machine, the results clearly show the promise of automated LF selection.

4.7 Accelerating the Search

In this section we discuss how we might improve the efficiency of our search within the LF model space.

One approach would be to avoid executing each candidate LF on the full set of HF samples, and instead use a partial execution to estimate its cost and correlation with the HF. To evaluate the potential of this approach we used our existing set of LF runs to compute the correlation of LFs on the Pareto front with the HF over partial samples (starting from only the first two and growing to the full 32). Figure 4.4 shows that this partial-sample estimate of HF-LF correlation has not converged until very close to the full sample, thus a partial evaluation is unlikely to be predictive of the behavior of an LF model in general.

If we have to fully execute an LF to properly gauge its efficiency, the only way to accelerate the search is to preemptively avoid evaluating as many inefficient LFs as possible. One way of achieving this is to devise some method of predicting how an arbitrary LF model might behave without actually running it (potentially using information from previous runs of similar models), then use this information to guide the search. For any simulation of sufficient complexity this prediction will necessarily be imprecise, so we may end up missing out on some efficient models. However, even with an imperfect oracle it may be possible to achieve a good tradeoff between acceleration and quality of results in the context of the full search: Even if we miss some of the models on the Pareto front, there will probably be many models near them that are almost as efficient, which would still leave us with a near-optimal selection.

In the rest of this section we discuss, based on the data from our exhaustive

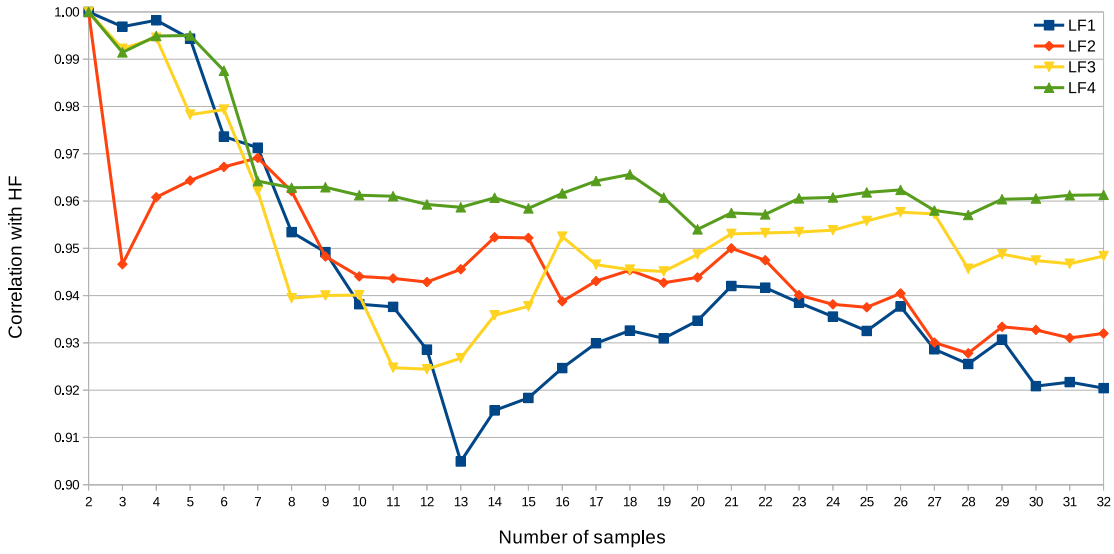

Figure 4.4: HF-correlation of Pareto-efficient LFs, computed on partial samples

LF search presented above, the potential of standard statistical modeling tools as predictors of LF model efficiency. We note that the conclusions we draw only apply to our specific setting and LF search space, and may not readily generalize to other situations; our goal is simply to evaluate the potential of the modeling methods we discuss, for which a proof-of-concept demonstration is sufficient.

The first (and simplest) approach we tried was to use a simple linear regression model to predict the efficiency of an LF model based on its selection of coarsening parameters. Specifically, we trained on different subsets of our exhaustive search data and tried to predict the rest. While a linear model was able to predict the cost of LFs with near-perfect accuracy using only 20% of the data for training, the same model was a poor fit to the LF correlation dataset, which only exhibited a correlation coefficient ρ of up to 0.3. In particular, the relationship between a model's level of fidelity and the correlation it achieves with the HF is not straightforward; for example, the best-correlated model within our search space was not actually the one with the least coarse choice for each parameter.

Even though it seems unlikely that we could build a perfect model for predicting LF correlation, it is still possible that we could build an imperfect model that would

still work to accelerate the search with minimal losses on average, as discussed above. To explore this idea further, we consider the simpler question of predicting whether an LF model will exhibit a correlation score higher than a given threshold. For this problem we will show that a simple linear classifier, trained using logistic regression and properly tuned, could indeed save us almost 75% of the work with only a 0.000747 average loss in correlation.

We use our set of ground truth data to emulate how such a classifier would be used in practice, and test how well different configurations of this classifier perform. We compare these different configurations in terms of average work saved and average correlation loss. In a real scenario we would only evaluate a subset of all LFs, train a classifier on the results of those runs, and of the remaining LFs only consider those that the classifier predicts will have correlation above the threshold. We evaluate this process by picking a random subset to train on from our exhaustive set of runs, and use the resulting model to predict the correlation of the remaining LFs. Any LFs we skip based on the decision of the classifier is considered saved work. If any of the LFs we skip has better correlation with the HF than the ones we ran on (either in the training set or the set of LFs accepted by the classifier), then the difference in correlation is considered a loss. Note that the loss in correlation depends partially on our initial split into training set and test set; we might get lucky and include the best-performing LF in our training set, and then even a bad classifier would have no correlation loss. For this reason it is important to run any evaluation of this method multiple times, with different random seeds, and report the average correlation loss.

Our classifier can be parameterized on two axes: its correlation threshold and the percentage of the dataset that we use to train it. The smaller the subset we train on the more work we save up-front, but smaller training sets will probably result in worse accuracy. Increasing the classifier's threshold will reduce the classifier's acceptance rate, thus reducing the number of LFs that the classifier suggests we run, and saving us more work. At the same time the false negative rate of the classifier will also increase, meaning more high-correlation LFs will probably be rejected, resulting in a higher potential for correlation loss.

In Figure 4.5 we plot the behavior of different classifier configurations in terms of

average work saved and average correlation loss. Each point represents the average performance of a different selection of classifier parameters across 100 runs. We tested every classifier configuration for classification threshold from 0.750 up to 0.995 (at intervals of 0.005) and for training set size from 5% up to 95% (at intervals of 5%). The results on this plot suggest that there is indeed a tradeoff between work saved and average correlation loss; it would be the user's choice how much of a loss they are prepared to accept. The configuration mentioned above, that saves us 74.67% of the work while only costing us 0.000747 in lost correlation, corresponds to a threshold of 0.925 and a training set size of 10%.

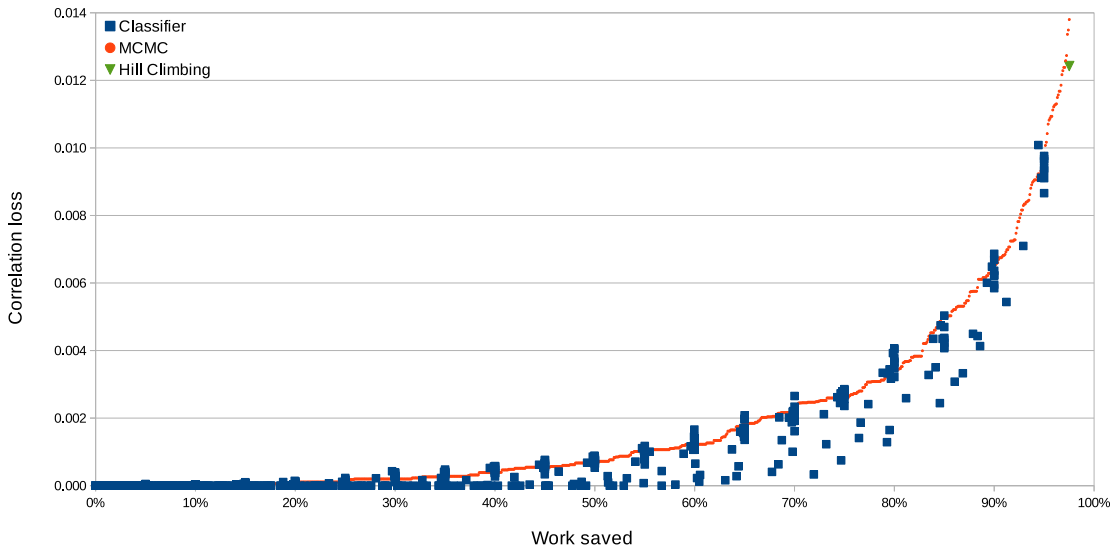

Figure 4.5: Results of LF search acceleration experiment

In the same figure we compare the efficiency of our classifier-based method with local search-based approaches. We performed 100 runs of a hill-climbing search, starting from a random LF each time. The average cost savings of this method was higher than any classifier configuration, but so was the loss in correlation. We also performed 100 randomly-initialized runs of a Markov chain Monte Carlo (MCMC) sampler following the Metropolis-Hastings algorithm [45, 31]. In the figure we plot the highest correlation that the different runs had discovered on average after exploring the same percentage of the search space (i.e. after they had performed the same

number of LF evaluations). The MCMC sampler appears to present a similar tradeoff between work saved and correlation lost as our classifier-based approach, but the latter can generally outperform it in how efficiently it trades off one for the other.

To gauge our method's sensitivity to changes in the classifier's parameters we compare the performance of the above efficient configuration with others that differ slightly in terms of training set size or classification threshold (see Table 4.3). From these numbers we observe that the system's overall behavior is reasonably robust to small changes.

adjustment	work saved	correlation loss
original	74.67%	0.000747
−0.005 classification threshold	67.74%	0.000407
+0.005 classification threshold	79.52%	0.001646
−5% training set size	76.46%	0.001410
+5% training set size	71.93%	0.000332

Table 4.3: Effect of adjustments to classifier parameters on overall performance, starting from the configuration with classification threshold 0.925 and a training set size of 10%.

4.8 Related Work

Most research in the area of Uncertainty Quantification focuses on developing good approximate models for different problems, or algorithms that use an existing set of models effectively [52, 24]. There has been much less work on our area of focus, i.e. selecting efficient combinations of coarsening options, balancing fidelity and computational cost. The most relevant prior work in the UQ literature includes methods that maintain a small set of predefined LFs, evaluate them all during each step of an outer-loop process and use the best one on each timestep [17, 47, 36], methods that pick an optimal LF by solving a non-linear optimization problem [43, 16, 40], multi-tier methods that optimize the evaluation ratio between the available LF levels [54], and methods that balance resource usage between improving a single LF model (exploration) and running it (exploitation) [53]. By their design, most of these

methods can only function over a small, predefined set of LFs, and/or only consider the fidelity of the LFs and not their cost. To our knowledge, the user study presented in this chapter is the first significant study of its kind.

Outside the field of Uncertainty Quantification, our approach is related to the technique of autotuning, which involves the systematic, feedback-driven exploration of a space of implementation choices, to identify the optimally-performing configuration (that is generally unique to the specific hardware environment). We refer the interested reader to the related work section of the Opentuner paper by Ansel et al [12]. Our setting is also similar to the hyperparameter tuning / model selection problem in Machine Learning, for which a number of approaches and tools are available, e.g. Spearmint [63], Hyperband [41] and Auto-WEKA [67].

Chapter 5

Optimizing Ensemble Execution

In this chapter we touch upon the subject of scheduling the execution of a UQ ensemble on a task-based system. Our goal in this discussion is to highlight the aspects of the problem that are unique to our setting, specifically the interaction between tiling and colocation, and the latency-throughput tradeoff. For this reason we restrict our focus to the simple case of a two-tier ensemble, for which the more traditional scheduling aspects of the problem have an easy solution. However, as we will show, even for this simplified case there are meaningful decisions to be made.

5.1 Definitions

We formally define our problem of interest as follows: Given a set of samples and a node allocation on a cluster, find the scheduling of those samples that minimizes total execution time.

The general problem as defined above is too broad to handle optimally. To have any hope of deriving an optimal algorithm we need to constrain our focus, by making certain simplifying assumptions:

- the nodes in our machine are equivalent
- the nodes in our machine are equidistant from each other (communication costs between any two nodes is the same)

- samples have (possibly) different computational requirements (in the case of Soleil-X, this accounts for differences in aspects like flow grid size, number of particles, number of simulated steps, physics models used)
- samples are independent (there is no cross-sample communication)
- samples can be arbitrarily tiled (i.e. split across nodes)
- each sample can be modeled as a loop that executes for a predefined number of iterations (this is typical for many simulation codes that involve a time-stepping loop)
- we only consider "offline" algorithms (i.e. our algorithm will run to completion before the ensemble starts, and make a decision about the entire execution of the ensemble)
- once started, samples cannot be preempted, stopped or moved
- a sample's execution starts and finishes on all allocated nodes at the same time
- once placed on a set of nodes, every iteration of the same sample will take the same amount of time (which is typical for many simulation codes, where each iteration of the time-stepping loop performs roughly the same computation); therefore, a sample's cost can be extrapolated by running it for a small number of iterations

Even with these simplifications the problem has an NP-hard core (it requires solving a multiprocessor scheduling problem [26]). Since our goal is not to provide a generic solution, but instead showcase the challenges unique to our setting, we choose to further restrict our focus to a case where an analytical solution exists.

Specifically, we will consider the case of a simple two-tier ensemble, with a single high-fidelity (HF) tier and a single low-fidelity (LF) tier, and where the following conditions hold:

- there are many more LFs than HFs

- we cannot fit more than one HF on a single node
- each LF can fit on a single node
- we have enough nodes in our allocation to run all the HFs in parallel
- the ensemble's running time is dominated by the HFs, so it does not make sense not to run them in parallel if we can

As a running example, we consider a two-tier ensemble for the case described in Section 4.2. For this discussion, all we need to know about this case is that it is comprised of two domains, "HIT" and "channel", where the first is used to generate inputs for the second. We use the HF presented in Section 4.2. As our LF we use LF4 from Table 4.2. We run on the same machine as that study (Lassen), utilizing only the GPUs for computation. For the rest of this chapter, when we refer to a "node" on Lassen we actually mean one GPU, of which each actual Lassen node contains four (for these experiments we instantiate a separate instance of the Legion runtime for each GPU).

Before we can compute the optimal solution to this scheduling problem, we need to measure the efficiency of the different possible ways of tiling a sample (splitting its execution across multiple nodes), and colocating samples (having multiple samples execute at the same time, rather than sequentially). We compare the different alternatives in terms of latency (the wall clock time required to complete a number of iterations, measured in seconds) and throughput (the normalized number of iterations completed per unit of computational resources, measured in iterations per second per node). There is generally a tradeoff between latency and throughput, so there will not necessarily be a single optimal way to tile or colocate. Instead, different choices will be better in different ensemble configurations, or even in different phases of the execution of the same ensemble.

For our example case we estimate this efficiency using partial runs: we run the different configurations for a small number of iterations and compute the above performance metrics. Because there is some inherent variance in a sample's performance, we run every trial multiple times and use the average value observed for each metric.

We run each trial at least 3 times, and make sure every one of the optimal configurations we output has a relative standard deviation[1] of at most 5% on the relevant metric. For these runs we disabled all output from the simulation.

5.2 HF Tiling

First we need to estimate, for every relevant node count, how efficient it is to strong-scale the HF to that number of nodes.

The smallest node count we need to consider is the minimum number of nodes that will fit an HF. As we scale to more nodes we expect the HF's latency to decrease but overall throughput to drop, and we expect to see diminishing returns in our latency gains. After some number of nodes the latency will stop decreasing, but throughput will continue to drop; it does not make sense to scale past that point. We may want to stop scaling even before that, if we reach a point where the nodes in our allocation are insufficient to run all HFs concurrently; in this case we would be forced to run two HFs on the same node one after the other and, given that the latency gains of strong scaling are sub-linear, this would cause overall latency to increase.

Similarly, it does not make sense to strong-scale two HFs differently. Say H_0 is strong-scaled more than H_1. Then either the ensemble's running time is dominated by H_1's latency (in which case it does not make any difference to the overall performance that we strong-scaled H_0 more), or neither HF's latency affects the critical path (in which case it makes sense to strong-scale both HFs at the same level as H_1, to increase throughput).

Depending on our target simulation, there may be more than one way to split an HF across a given number of nodes. We would then want to pick, for each node count, the configuration with the highest throughput.

Table 5.1 summarizes the results of the HF tiling measurements for our example case: We run 5000 iterations of the HF under different tiling configurations and report on the observed latency and throughput. We start our measurements at the smallest

[1] The relative standard deviation of a statistical sample is defined as the sample standard deviation divided by the sample mean.

number of nodes that can fit an HF (a single node on Lassen).

#Nodes	Channel placed on nodes	Latency (s)	Throughput (iters/s/node)
1	1	868.443	**5.7574**
2	1,2	806.697	3.0991
3	1,2,3	1088.167	1.5316
4	1,2,3,4	1382.833	0.9039
2	2	516.679	**4.8386**
3	2,3	888.135	1.8766
4	2,3,4	1117.317	1.1188

Table 5.1: Evaluation of HF tiling options (running one HF for 5000 iterations, with the HIT domain always placed on node 1)

In this simulation, each of the two domains can be split independently, so that the same number of nodes can be utilized in different ways. Because the HIT domain is a fourth the size of the channel domain, it makes sense to start with splitting the latter before we start splitting the former. All the configurations in Table 5.1 table have the HIT domain running on a single node (always node 1); we reach the point of diminishing returns before we would need to consider how to split the HIT domain.

This experiment shows that the HF exhibits typical strong scaling behavior: The one possible 1-node configuration achieves a throughput of 5.7574 iters/s/node. The optimal 2-node configuration (which involves separating the two domains across nodes) improves on the latency (falling short of actually halving it), but achieves a lower throughput of 4.8385 iters/s/node. Further splitting does not reduce latency, but achieves even lower throughput; therefore from this point on we will only consider scaling the HF on up to 2 nodes.

5.3 HF-HF Colocation

We have assumed it is impossible to fit more than one HF on a node (and one node may not even be enough). On Lassen we can fit exactly one HF per node, because each HF uses around 78% of the node's GPU memory.

It does not make sense to colocate strong-scaled HFs. For example, say we have 2 HFs, H_0 and H_1 (that can fit on 1 node each), each strong-scaled across two nodes, N_a

and N_b. We could instead assign H_0 exclusively to N_a and H_1 exclusively to N_b; then both samples would execute at a higher throughput, and thus both would terminate faster.

5.4 LF Tiling

It does not make sense to strong-scale LFs on more than 1 node: We have assumed that it is the latency of the HFs that dominates the running time of the ensemble; LFs complete much faster than HFs so their latency is not a concern, instead we care only about maximizing their throughput, and splitting a sample results in decreased throughput.

5.5 LF-LF Colocation

Next we need to identify the optimal number of LFs to run concurrently on a single node. To do this for our example case we ran multiple short ensembles of only LFs, each LF running for 5000 iterations, and all LFs occupying the same node. We report our measurements in Table 5.2. As explained in Section 5.4, we only compare the efficiency of LF-only combinations in terms of throughput.

As expected, running only one LF at a time is not the optimal configuration. This happens because LFs typically have small domain sizes, resulting in small workloads for GPU kernels, and thus not enough computation to effectively hide the latency of other parts of the application (e.g. the communication, runtime analysis, or GPU kernel launching).

As more LFs are executed concurrently, both GPU utilization and overall throughput increases, until we hit the limit of how many LFs can fit on a node (which did not happen during the above experiment, because each LF used only around 3% of a Lassen node's GPU memory), or we saturate some resource available to the application (e.g. the GPU's processing capacity, the CPU cores available to the runtime, or the communication channel), which seems to happen at 2 LFs per node for our example case.

#LFs	Latency (s)	Throughput (iters/s/node)
1	160.820	31.0907
2	270.077	**37.0265**
3	427.795	35.0635
4	577.274	34.6456
5	743.052	33.6450
6	914.295	32.8122
7	1041.611	33.6018
8	1191.949	33.5585
9	1356.078	33.1839
10	1495.749	33.4281
11	1639.876	33.5391
12	1789.157	33.5353
13	1909.598	34.0386
14	2048.713	34.1678
15	2195.111	34.1668
16	2326.518	34.3862

Table 5.2: Evaluation of LF-LF colocation options (running multiple LFs on a single node, each for 5000 iterations)

We have assumed there are many more LFs than HFs, so we can assume we can always create groups of LFs of the most efficient size, to run concurrently. Therefore, in subsequent sections we assume we can always achieve the maximum throughput (of 37.0265 iters/s/node for our example case) when running LFs on their own.

5.6 HF-LF Colocation

Finally, we need to identify if there is any efficiency to be gained by running one or more LFs concurrently with an HF, and what the best combination is (the answer may be different for different tilings of the HF).

The rationale behind this question is that, even when executing an HF, not all of the available computational resources are used at their full capacity. Therefore, by running some additional low-intensity work on the same resources at the same time, we may be able to achieve better overall utilization.

For the purposes of this discussion we will not attempt to tune the joint execution

of the HF and LFs, and will just let the runtime decide how to allocate resources among the samples running on the same nodes. Therefore, we expect that the more LFs we colocate with the HF, the more resources will be allocated to the LFs over the HF, thus the LF aggregate throughput will rise at the expense of the HF throughput. We then need a way to decide, of all the available colocation options, which is the best tradeoff for our purposes.

Say we are given an HF-LF colocation configuration and we know that, while running under this configuration, the HF achieves a throughput of T_H^c and the LFs a throughput of T_L^c. Say also that the corresponding throughput numbers are T_H^s and T_L^s if running separately (for the LFs we would use the throughput achieved in the optimal LF-only colocation configuration). Based on these figures, we will now develop a method to evaluate how much more efficient (if at all) it is to colocate an HF with LFs under the given configuration, over running them separately.

Without loss of generality say we are executing on 1 node. Say we wanted to run I iterations across all LFs. This would take time $t^a = I/T_L^s$ if we were to run the LFs separately from the HF. Instead say we colocated the LFs with the HF until all LF iterations are completed, which would take time $t = I/T_L^c$. For that period of time the HF will run at the (reduced) throughput of T_H^c instead of T_H^s, therefore it will complete $(T_H^s - T_H^c)t = (T_H^s - T_H^c)I/T_L^c$ fewer iterations, which will require an extra time of $t^c = (T_H^s - T_H^c)I/(T_L^c T_H^s)$ to run (at full HF throughput). If $t^a > t^c$ then we saved time by running the LF colocated with the HF. We can thus use the ratio $r = t^a/t^c = (IT_L^c T_H^s)/(T_L^s(T_H^s - T_H^c)I) = (T_L^c T_H^s)/(T_L^s(T_H^s - T_H^c))$ to quantify how much more efficient it is to run in an HF-LF colocation configuration over running separately (colocation is more efficient iff this ratio is larger than 1).

All we need to do now to evaluate the different HF-LF colocation options for our example case is to estimate the values of T_H^c and T_L^c. We already know that $T_L^s = 37.0265$ iters/s/node, $T_H^s = 5.7574$ iters/s/node on 1 node and $T_H^s = 4.8386$ iters/s/node on 2 nodes (we only consider the best HF tiling configuration for each node count). To do this we run the different configurations for a small duration (5000 iterations for every sample), pick a time point into the execution when all samples were still running (the same point for all configurations) and count how many iterations

the HF and the LFs in aggregate had completed at that point. We use this number to compute the throughput values and calculate the ratio r.

Table 5.3 summarizes our measurements for all possible HF-LF colocation configurations on 1 node. We took our measurements at 500s into the execution. We could not colocate more than 7 LFs with the HF, because that would require more GPU memory than is available on a Lassen node. We note that initially the colocation increases overall efficiency, but eventually overall performance starts to drop. The highest efficiency is achieved when colocating 3 LFs with the HF; in this configuration the HF executes at a throughput of 4.3480 iters/s/node, and the LFs at 18.8633 iters/s/node.

#LFs	Iters completed		Throughput		Ratio
	HF	LFs	HF	LFs	
1	2459	3722	4.9183	7.4431	1.38
2	2316	6891	4.6320	13.7814	1.90
3	2174	9432	**4.3480**	**18.8633**	**2.08**
4	2004	10626	4.0073	21.2513	1.89
5	1668	11083	3.3355	22.1663	1.42
6	1484	11839	2.9686	23.6770	1.32
7	1294	12025	2.5873	24.0500	1.18

Table 5.3: Evaluation of 1-node HF-LF colocation options (multiple LFs colocated with 1 HF, each running 5000 iterations, measuring after 500s of execution)

Table 5.4 covers our experiments with 2-node HF-LF colocation configurations. We took our measurements at 200s into the execution. It was possible to execute more than 4 LFs per node, but based on the r ratio's trend it seemed unlikely that larger colocation counts would beat the performance of the optimal configuration, i.e. a single LF on node 1 (under this, the HF executes at 4.8142 iters/s/node and the LFs at 7.9058 iters/s/node). We note that the value of r for this case is significantly larger than the others because the throughput of the HF is hardly affected by the presence of the LF (it only decreases from 4.8385 iters/s/node to 4.8142 iters/s/node), suggesting that node 1 is significantly underutilized running just the HIT domain.

#LFs placed on		Iters completed		Throughput		Ratio
Node 1	Node 2	HF	LFs	HF	LFs	
0	1	1645	1253	4.1125	3.1317	0.56
0	2	1546	2193	3.8650	5.4817	0.74
0	3	1498	3081	3.7458	7.7017	0.92
0	4	1414	3787	3.5358	9.4667	0.95
1	0	1926	3162	**4.8142**	**7.9058**	**42.46**
1	1	1618	4957	4.0458	12.3933	2.04
1	2	1532	6248	3.8300	15.6208	2.02
1	3	1510	6810	3.7750	17.0242	2.09
1	4	1268	8373	3.1692	20.9325	1.64
2	0	1487	3940	3.7175	9.8492	1.15
2	1	1542	5665	3.8542	14.1625	1.88
2	2	1488	6570	3.7208	16.4250	1.92
2	3	1329	7902	3.3233	19.7558	1.70
2	4	1092	9407	2.7300	23.5167	1.46
3	0	1122	4486	2.8058	11.2150	0.72
3	1	1143	7475	2.8583	18.6883	1.23
3	2	1133	8641	2.8317	21.6025	1.41
3	3	1115	9209	2.7883	23.0217	1.47
3	4	993	9562	2.4833	23.9050	1.33
4	0	880	4695	2.2008	11.7367	0.58
4	1	939	8740	2.3483	21.8492	1.15
4	2	917	10062	2.2925	25.1550	1.29
4	3	894	9982	2.2358	24.9550	1.25
4	4	871	9761	2.1767	24.4017	1.20

Table 5.4: Evaluation of 2-node HF-LF colocation options (multiple LFs colocated with 1 HF, each running 5000 iterations, measuring after 200s of execution)

5.7 Scheduling Algorithm

At this point we have enough information to derive an optimal schedule for the execution of our ensemble. We will present our scheduling algorithm through an example: an ensemble following our example case configuration comprised of 1 HF and 96 LFs, to be executed on 4 Lassen nodes. It becomes relevant at this point to know how many iterations each sample needs to run for: the HF needs to complete 2,365,703 iterations, and each LF 295,713 iterations.

For now we will ignore the fact that LFs need to be scheduled as units of work, and instead assume we can schedule them at the level of an iteration; this will simplify our algorithm but is unrealistic, so our final solution will need to be adjusted.

We first consider, for each HF tiling configuration (starting from the minimum amount of tiling), the schedule that involves running HFs and LFs on separate nodes.

Our first option is to use 1 node for the HF (see Figure 5.1). In this configuration we are "latency-limited": The overall runtime is dominated by the latency of the HF. All the LFs complete before the HF and the nodes allocated to them remain idle until the HF finishes. Therefore it makes sense to try tiling the HF further, trading off some throughput for reduced latency (we are wasting resources anyway by leaving nodes idle). We continue tiling for as long as we remain latency-limited, there are enough nodes available to tile further, and tiling continues to reduce latency. If we exhaust all possible HF tiling options and are still latency-limited, then the most-tiled configuration is the optimal solution.

We continue with tiling the HF to 2 nodes (see Figure 5.2). We have now reached the point where we are "throughput-limited": The HF finishes before all the LFs, so the HF's latency no longer matters and throughput is the only bottleneck. This suggests we do not need to consider any further tiling of the HF, as that would only serve to reduce the HF's throughput.

We can further improve the efficiency of the last schedule. As a first step, we notice that the HF's nodes will be idle towards the end of the run, so we reschedule LFs to fill that time (see Figure 5.3).

If an efficient HF-LF colocation configuration is available (one with ratio $r > 1$,

total number of HF iterations:

$$I_H = 2,365,703 \qquad iterations$$

total number of LF iterations:

$$I_L = 96 \cdot 295,713 = 28,388,448 \qquad iterations$$

nodes used for the HF:

$$N_H = 1 \qquad nodes$$

nodes used for the LFs:

$$N_L = 3 \qquad nodes$$

HF throughput when running separately:

$$T_H^s = 5.7574 \qquad iters/s/node$$

LF throughput when running separately:

$$T_L^s = 37.0265 \qquad iters/s/node$$

total running time for the HF:

$$t_1 = I_H/(T_H^s N_H) = 410,898 \qquad seconds$$

total running time for the LFs:

$$t_2 = I_L/(T_L^s N_L) = 255,569 \qquad seconds$$

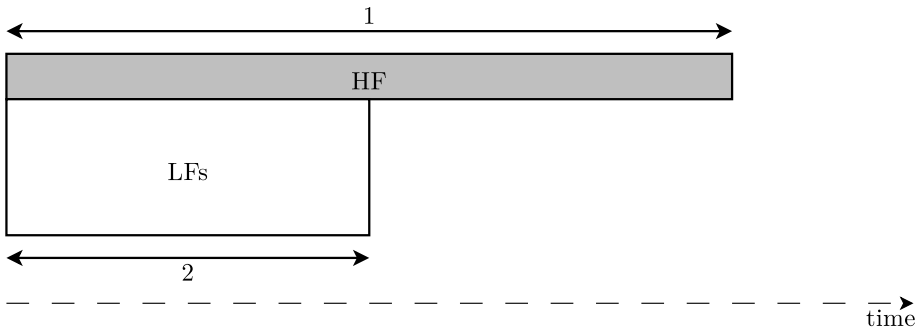

Figure 5.1: Scheduling tiers separately, HF on 1 node

total number of HF iterations:

$$I_H = 2,365,703 \qquad iterations$$

total number of LF iterations:

$$I_L = 96 \cdot 295,713 = 28,388,448 \qquad iterations$$

nodes used for the HF:

$$N_H = 2 \qquad nodes$$

nodes used for the LFs:

$$N_L = 2 \qquad nodes$$

HF throughput when running separately:

$$T_H^s = 4.8386 \qquad iters/s/node$$

LF throughput when running separately:

$$T_L^s = 37.0265 \qquad iters/s/node$$

total running time for the HF:

$$t_1 = I_H/(T_H^s N_H) = 244,462 \qquad seconds$$

total running time for the LFs:

$$t_2 = I_L/(T_L^s N_L) = 383,353 \qquad seconds$$

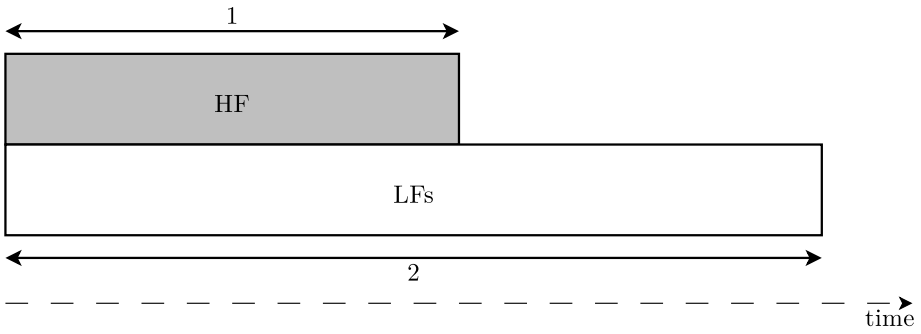

Figure 5.2: Scheduling tiers separately, HF on 2 nodes

total number of HF iterations:

$$I_H = 2,365,703 \qquad iterations$$

total number of LF iterations:

$$I_L = 96 \cdot 295,713 = 28,388,448 \qquad iterations$$

nodes used for the HF:

$$N_H = 2 \qquad nodes$$

nodes used for the LFs (initially):

$$N_L = 2 \qquad nodes$$

HF throughput when running separately:

$$T_H^s = 4.8386 \qquad iters/s/node$$

LF throughput when running separately:

$$T_L^s = 37.0265 \qquad iters/s/node$$

total running time for the HF:

$$t_1 = I_H/(T_H^s N_H) = 244,462 \qquad seconds$$

In that time the LFs have completed:

$$I_1 = T_L^s t_1 N_L = 18,103,144 \qquad iterations$$

The remaining LF iterations will use all nodes, taking:

$$t_2 = (I_L - I_1)/T_L^s/(N_H + N_L) = 69,446 \qquad seconds$$

total execution time:

$$t_{all} = t_1 + t_2 = 313,908 \qquad seconds$$

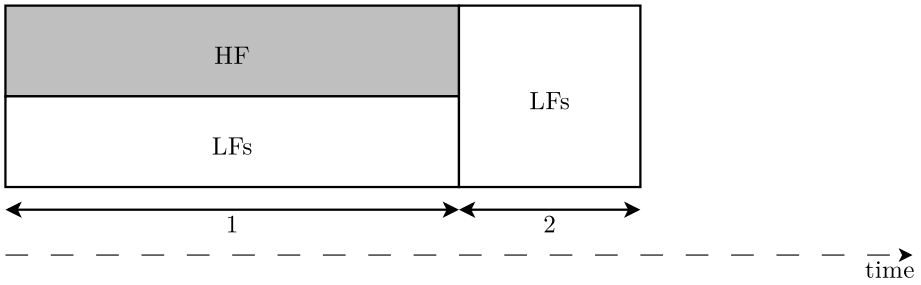

Figure 5.3: Scheduling tiers separately, HF on 2 nodes, eliminating idle time

see Section 5.6), we can use it to colocate some LFs with the HF, thus trading back some latency (which is no longer an issue) for throughput. If our ensemble is such that the execution of the HF can cover the entire "tail" of LFs, then we would solve the equations in Figure 5.4 to decide how many LF iterations to colocate.

Say we colocate I_L^1 LF iterations with the HF; these will run in:

$$t_1 = I_L^1/(T_L^c N_H) \qquad seconds$$

Over that period of time the HF will complete:

$$I_H^1 = T_H^c N_H t_1 = T_H^c I_L^1 / T_L^c \qquad iterations$$

The remaining HF iterations will execute by themselves, and take:

$$t_2 = (I_H - I_H^1)/(T_H^s N_H) \qquad seconds$$

The rest of the LF iterations run on their own nodes, and take:

$$t_3 = (I_L - I_L^1)/(T_L^s N_L) \qquad seconds$$

In the optimal configuration, the HF and remaining LFs finish together:

$$t_3 = t_1 + t_2$$

We solve the last equation to get I_L^1.

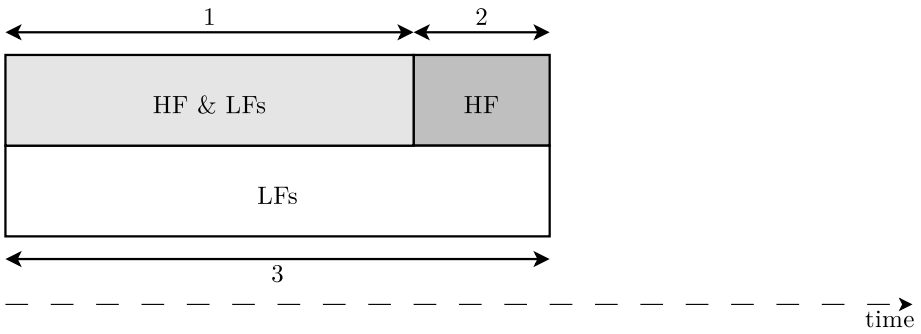

Figure 5.4: Colocating tiers, HF on 2 nodes, long HF case

If, on the other hand, we can colocate LFs for the entire duration of the HF's execution and still have LF work left over (as happens for our example), we schedule as in Figure 5.5. We know that we need to follow this approach if, while trying to apply the formula for I_L^1 from the previous case, we end up with $I_H^1 > I_H$.

total number of HF iterations:

$$I_H = 2,365,703 \qquad iterations$$

total number of LF iterations:

$$I_L = 96 \cdot 295,713 = 28,388,448 \qquad iterations$$

nodes used for the HF:

$$N_H = 2 \qquad nodes$$

nodes used exclusively for the LFs:

$$N_L = 2 \qquad nodes$$

throughput when running separately:

$$T_H^s = 4.8386 \qquad iters/s/node$$

$$T_L^s = 37.0265 \qquad iters/s/node$$

throughput when running colocated:

$$T_H^c = 4.8142 \qquad iters/s/node$$

$$T_L^c = 7.9058 \qquad iters/s/node$$

Running fully in colocated mode, the HF will take:

$$t_1 = I_H/(T_H^c N_H) = 245,701 \qquad seconds$$

In that time the LFs colocated with the HF will complete:

$$I_L^{1c} = T_L^c t_1 N_H = 3,884,926 \qquad iterations$$

The LFs running by themselves on the rest of the nodes will complete:

$$I_L^{1s} = T_L^s t_1 N_L = 18,194,896 \qquad iterations$$

The remaining LF iterations will use all nodes, taking:

$$t_2 = (I_L - I_L^{1c} - I_L^{1s})/T_L^s/(N_H + N_L) = 42,595 \qquad seconds$$

total execution time:

$$t_{all} = t_1 + t_2 = 288,296 \qquad seconds$$

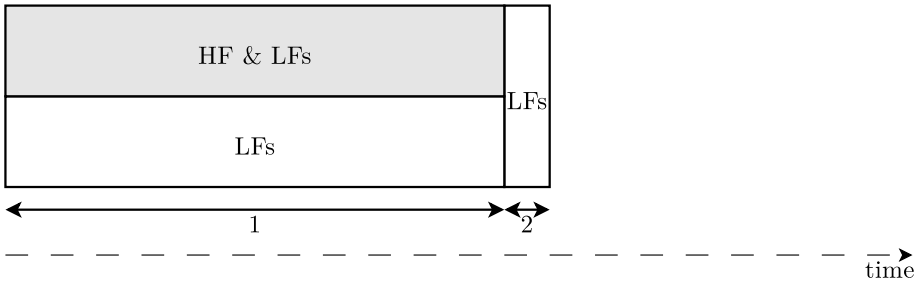

Figure 5.5: Colocating tiers, HF on 2 nodes, short HF case

The optimal schedule will be either the least latency-limited configuration we considered or the throughput-limited one (including all optimizations). For our example case the throughput-limited configuration from Figure 5.5 is optimal, requiring a total of 288,296 seconds to complete (over the latency-limited configuration's 410,898 seconds, as computed in Figure 5.1). The full scheduling algorithm is provided in Figure 5.6.

As mentioned previously, this analysis assumes we can schedule at the level of an LF iteration. Therefore the optimal schedule may be infeasible, for a number of reasons:

- The optimal number of LF iterations to colocate with the HF may not be an exact multiple of a sample's size.
- The end of the HF's execution may not line up exactly with the end of the standalone LFs' execution.
- The number of LFs remaining after the HF's execution finishes may not be appropriate for scheduling in the optimal LF colocation configuration.

We therefore need to adjust the optimal (but infeasible) schedule into the closest feasible one (that may be slightly less efficient).

5.8 Related and Future Work

The problem studied in this chapter falls under the category of scheduling problems, a well-studied class of problems in Computer Science with multiple variations, most of them NP-hard. Our specific case is an instance of parallel task scheduling [20, 60], and is closest to the moldable task assumption [72]. Under this assumption, different tasks (in our case samples) can be split independently from each other (with progressively lower performance gains at higher degrees of splitting), and this splitting remains constant for the duration of their execution. This problem is NP-hard in the general case, but efficient approximate algorithms exist [22].

Input: I_H: total number of HF iterations
I_L: total number of LF iterations
N: total number of nodes
N_H^{min}: minimum number of nodes that will fit an HF
N_H^{max}: strong-scaling limit for HF
$T_H^s[n]$: HF throughput when running separately, on n nodes
T_L^s: LF throughput when running separately
$T_H^c[n]$: HF throughput when running colocated, on n nodes
$T_L^c[n]$: LF throughput when running colocated, on n nodes
Output: the running time of the optimal schedule

$t_{lat} = \infty$
for $N_H = N_H^{min}$ **to** ∞ **do**
  if $N_H > N$ **or** $N_H > N_H^{max}$ **then**
    return t_{lat}
  $N_L = N - N_H$
  if $N_L = 0$ **then**
    break
  $t_1 = I_H/(T_H^s[N_H]N_H)$
  $t_2 = I_L/(T_L^s N_L)$
  if $t_1 > t_2$ **then**
    $t_{lat} = t_1$
  else
    break
solve for I_H^1, I_L^1, t_{tput} **:**
  $t_{tput} = I_L^1/(T_L^c[N_H]N_H) + (I_H - I_H^1)/(T_H^s[N_H]N_H)$
  $I_L - I_L^1 = T_L^s t_{tput} N_L$
  $I_H^1 = T_H^c[N_H] I_L^1 / T_L^c[N_H]$
if $I_H^1 \leq I_H$ **then**
  return $\texttt{min}(t_{lat}, t_{tput})$
$t_1 = I_H/(T_H^c[N_H]N_H)$
$I_L^{1c} = T_L^c[N_H] t_1 N_H$
$I_L^{1s} = T_L^s t_1 N_L$
$t_2 = (I_L - I_L^{1c} - I_L^{1s})/(T_L^s N)$
$t_{tput} = t_1 + t_2$
return $\texttt{min}(t_{lat}, t_{tput})$

Figure 5.6: Full scheduling algorithm

The discussion in this chapter is concerned with two-tier ensembles, a variant of the problem that admits an analytical solution. Future work on handling ensembles of three or more tiers would need to solve the full scheduling problem. Unfortunately, the algorithms proposed in prior work are not immediately applicable to our setting, as they do not consider the effect of colocation, and would need to be adapted. Additionally, increasing the number of tiers would cause a combinatorial increase in the number of possible tiling and colocation configurations to consider, so we might have to make further simplifying assumptions, to keep the number of combinations under control.

A particularly promising direction for future work involves scheduling across both the CPUs and the GPUs on every node, thus utilizing the entire set of available hardware. To extend our scheduling algorithm to this setting, we would need to answer a number of additional questions:

- How should we split CPU cores between the runtime and running samples? Runtime overhead is a major component in the efficiency of different configurations, so we need to ensure that sufficient resources are allocated to the runtime analysis. Additionally, it may be profitable to split the application's cores across multiple OpenMP thread pools, to better handle NUMA effects.

- Should we consider splitting a single simulation between two types of processors? As outlined in Section 2.8, this is not the case for Soleil-X.

- Should we consider scheduling HFs on CPUs (and if so, how far is it profitable to tile them across nodes)? On Lassen the HF executes 9x slower using 32 out of a node's 40 cores (with 8 allocated to the runtime) over OpenMP, compared to running on a GPU. Therefore, any HF placed on the CPUs would significantly dominate the running time of the ensemble.

- What is the optimal throughput we can achieve for LFs running on CPUs? We would need to consider different colocation configurations, similar to the analysis in Section 5.5.

- Should we consider tiling LFs across two nodes' CPUs? For reasons similar to those outlined in Section 5.4 we do not need to consider this option.
- How does the presence of LFs on CPUs affect the throughput of different execution configurations on GPUs? We would need to redo our throughput measurements with the CPUs occupied, and even under different CPU occupancy configurations. Ultimately the decision of how to schedule on the CPUs becomes an additional parameter to consider when deriving an optimal-throughput configuration.

To handle the more general problem of scheduling ensembles with three or more tiers on both CPUs and GPUs we would need to explore parallel scheduling algorithms for the heterogeneous setting [68].

Further extensions to the kinds of ensembles we handle can correspondingly exploit existing work. For example, to support dynamically-updated ensembles we would rely on algorithms for the online setting, and to support dynamic re-partitioning of running samples we would research scheduling algorithms for the malleable (instead of moldable) task assumption.

Chapter 6

Conclusion

In this dissertation we explored the potential of task-based programming systems to improve the practice of building complex multi-physics simulations, and performing Uncertainty Quantification studies over them.

We reported on our experience building a multi-physics solver, Soleil-X, in the Legion task-based programming system, and extending it to support UQ ensembles. We discussed our major design choices, particularly the co-design of distribution and domain coupling strategies, and our methodology for optimizing the application. To evaluate our solver's scalability we performed a weak-scaling study on Sierra, a leadership-class supercomputer, where we achieved good scaling up to 256 nodes.

We explored in depth one of the ways that task-based runtimes can be used to automate the practice of performing UQ studies. Specifically, we developed a search-based approach for selecting the optimal low-fidelity model for a UQ study. We built a prototype implementation of this approach and applied it to a medium-size simulation, where we were able to outperform the choices of human experts. We reported on some of the issues we faced while making the base simulation robust enough to be able to handle the wide variety of configurations explored during our search. We also discussed some promising preliminary ideas for accelerating our search process using statistical modeling.

Finally, we discussed the problem of optimizing the execution of UQ ensembles. We focused on the simple but common case of a two-tier ensemble, that highlights the

unique aspects of our setting while admitting an analytical solution. We developed a method for solving this problem that starts by finding the optimal configurations of tiling and colocation between the two fidelities, then combines this information into a deterministic scheduling algorithm.

Bibliography

[1] Exascale Computing Engineering Center. Predictive Science Academic Alliance Program (PSAAP) II, Stanford University. http://exascale.stanford.edu.

[2] Lassen, Lawrence Livermore National Laboratory. https://computing.llnl.gov/computers/lassen.

[3] Piz Daint, Swiss National Supercomputing Centre. https://www.cscs.ch/computers/piz-daint/.

[4] Sierra, Lawrence Livermore National Laboratory. https://computing.llnl.gov/computers/sierra.

[5] Summit, Oak Ridge Leadership Computing Facility. https://www.olcf.ornl.gov/summit/.

[6] Titan, Oak Ridge Leadership Computing Facility. https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/.

[7] Brian M. Adams, W.J. Bohnhoff, K.R. Dalbey, J.P. Eddy, M.S. Eldred, D.M. Gay, K. Haskell, Patricia D. Hough, and Laura P. Swiler. DAKOTA, a Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 5.0 Users Manual. *Sandia National Laboratories, Tech. Rep. SAND2010-2183*, 2009.

[8] Michael P. Adams, Marvin L. Adams, W. Daryl Hawkins, Timmie Smith, Lawrence Rauchwerger, Nancy M. Amato, Teresa S. Bailey, and Robert D. Falgout. Provably Optimal Parallel Transport Sweeps on Regular Grids. Technical report, Lawrence Livermore National Lab (LLNL), Livermore, CA, USA, 2013.

[9] Alex Aiken, Michael Bauer, and Sean Treichler. Realm: An Event-Based Low-Level Runtime for Distributed Memory Architectures. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 263–275. IEEE, 2014.

[10] Juan Alonso, Seonghyeon Hahn, Frank Ham, Marcus Herrmann, Gianluca Iaccarino, Georgi Kalitzin, Patrick LeGresley, Ken Mattsson, Gorazd Medic, Parviz Moin, et al. CHIMPS: A High-Performance Scalable Module for Multi-Physics Simulations. In *42nd AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit*, page 5274, 2006.

[11] Michael J. Andrews and Peter J. O'Rourke. The Multiphase Particle-In-Cell (MP-PIC) Method for Dense Particulate Flows. *International Journal of Multiphase Flow*, 22(2):379–402, 1996.

[12] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316. ACM, 2014.

[13] Maxime Bassenne, Javier Urzay, George I. Park, and Parviz Moin. Constant-Energetics Physical-Space Forcing Methods for Improved Convergence to Homogeneous-Isotropic Turbulence with Application to Particle-Laden Flows. *Physics of Fluids*, 28(3):035114, 2016.

[14] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage*

and Analysis, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[15] Dan Bonachea and Paul H. Hargrove. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. Technical Report LBNL-2001174, Lawrence Berkeley National Laboratory, October 2018. Languages and Compilers for Parallel Computing (LCPC'18).

[16] Souma Chowdhury, Ali Mehmani, and Achille Messac. Concurrent Surrogate Model Selection (COSMOS) Based on Predictive Estimation of Model Fidelity. In *ASME 2014 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages V02BT03A026–V02BT03A026. Citeseer, 2014.

[17] Ivo Couckuyt, Filip De Turck, Tom Dhaene, and Dirk Gorissen. Automatic Surrogate Model Type Selection During the Optimization of Expensive Black-Box Problems. In *Proceedings of the Winter Simulation Conference*, pages 4274–4284. Winter Simulation Conference, 2011.

[18] Clayton T. Crowe, John D. Schwarzkopf, Martin Sommerfeld, and Yutaka Tsuji. *Multiphase Flows with Droplets and Particles*. CRC Press, second edition, 2011.

[19] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[20] Eliezer Dekel and Sartaj Sahni. Parallel Scheduling Algorithms. *Operations Research*, 31(1):24–49, 1983.

[21] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A Multi-Stage Language for High-Performance Computing. *ACM SIGPLAN Notices*, 48(6):105–116, 2013.

[22] Pierre-François Dutot, Grégory Mounié, and Denis Trystram. Scheduling Parallel Tasks: Approximation Algorithms, 2004.

[23] Martha W. Evans, Francis H. Harlow, and Eleazer Bromberg. The Particle-In-Cell Method for Hydrodynamic Calculations. Technical report, Los Alamos National Lab (NM), 1957.

[24] M Giselle Fernández-Godino, Chanyoung Park, Nam-Ho Kim, and Raphael T Haftka. Review of Multi-Fidelity Models. *arXiv preprint arXiv:1609.07196*, 2016.

[25] Joel H. Ferziger and Milovan Perić. *Computational Methods for Fluid Dynamics*. Springer, Berlin, third edition, 2002.

[26] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, page 238. W. H. Freeman, San Francisco, 1979.

[27] J. Davison de St Germain, John McCorquodale, Steven G. Parker, and Christopher R. Johnson. Uintah: A Massively Parallel Problem Solving Environment. In *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, pages 33–41. IEEE, 2000.

[28] Panagiotis E. Hadjidoukas, Panagiotis Angelikopoulos, Costas Papadimitriou, and Petros Koumoutsakos. Π4U: A High Performance Computing Framework for Bayesian Uncertainty Quantification of Complex Models. *Journal of Computational Physics*, 284:1–21, 2015.

[29] Panagiotis E. Hadjidoukas, Evaggelos Lappas, and Vassilios V. Dimakopoulos. A Runtime Library for Platform-Independent Task Parallelism. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 229–236. IEEE, 2012.

[30] J. M. Hammersley and D. C. Handscomb. *Monte Carlo Methods*. Springer Netherlands, first edition, 1964.

[31] W. Keith Hastings. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. 1970.

[32] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua – An Extensible Extension Language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[33] Xiangmin Jiao, Gengbin Zheng, Phillip A. Alexander, Michael T. Campbell, Orion S. Lawlor, John Norris, Andreas Haselbacher, and Michael T. Heath. A System Integration Framework for Coupled Multiphysics Simulations. *Engineering with Computers*, 22(3-4):293–309, 2006.

[34] Lluis Jofre, Gianluca Geraci, Hillary Fairbanks, Alireza Doostan, and Gianluca Iaccarino. Multi-Fidelity Uncertainty Quantification of Irradiated Particle-Laden Turbulence. *arXiv preprint arXiv:1801.06062*, 2018.

[35] David E. Keyes, Lois C. McInnes, Carol Woodward, William Gropp, Eric Myra, Michael Pernice, John Bell, Jed Brown, Alain Clo, Jeffrey Connors, et al. Multiphysics Simulations: Challenges and Opportunities. *The International Journal of High Performance Computing Applications*, 27(1):4–83, 2013.

[36] Slawomir Koziel and Adrian Bekasiewicz. Low-Cost Multiband Compact Branch-Line Coupler Design Using Response Features and Automated EM Model Fidelity Adjustment. *International Journal of RF and Microwave Computer-Aided Engineering*, 28(4):e21233, 2018.

[37] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[38] Wonchan Lee, Manolis Papadakis, Elliott Slaughter, and Alex Aiken. A Constraint-Based Approach to Automatic Data Partitioning for Distributed Memory Execution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, 2019.

[39] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. Dynamic Tracing: Memoization of Task

Graphs for Dynamic Task-Based Runtimes. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 441–453. IEEE, 2018.

[40] Leifur Leifsson, Slawomir Koziel, and Piotr Kurgan. Automated Low-Fidelity Model Setup for Surrogate-Based Aerodynamic Optimization. In *Solving Computationally Expensive Engineering Problems*, pages 87–111. Springer, 2014.

[41] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[42] Stefano Marelli and Bruno Sudret. UQLab: A Framework for Uncertainty Quantification in Matlab. In *Vulnerability, Uncertainty, and Risk: Quantification, Mitigation, and Management*, pages 2554–2563. 2014.

[43] Ali Mehmani, Souma Chowdhury, Jie Zhang, and Achille Messac. A Novel Approach to Simultaneous Selection of Surrogate Models, Constitutive Kernels, and Hyper-Parameter Values. In *10th AIAA Multidisciplinary Design Optimization Conference*, page 1487, 2014.

[44] Qingyu Meng, Justin Luitjens, and Martin Berzins. Dynamic Task Scheduling for the Uintah Framework. In *2010 3rd Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–10. IEEE, 2010.

[45] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[46] Michael F. Modest. *Radiative Heat Transfer*. Academic Press, third edition, 2013.

[47] Joshua Mullins and Sankaran Mahadevan. Variable-Fidelity Model Selection for Stochastic Simulation. *Reliability Engineering & System Safety*, 131:40–52, 2014.

[48] Leo W. T. Ng and Karen E. Willcox. Multifidelity Approaches for Optimization Under Uncertainty. *International Journal for Numerical Methods in Engineering*, 100(10):746–772, 2014.

[49] John Nickolls, Ian Buck, and Michael Garland. Scalable Parallel Programming. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 40–53. IEEE, 2008.

[50] Francisco Palacios, Juan Alonso, Karthikeyan Duraisamy, Michael Colonno, Jason Hicken, Aniket Aranake, Alejandro Campos, Sean Copeland, Thomas Economon, Amrita Lonkar, et al. Stanford University Unstructured (SU^2): An Open-Source Integrated Computational Environment for Multi-Physics Simulation and Design. In *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, page 287, 2013.

[51] Raghu Pasupathy, Bruce W. Schmeiser, Michael R. Taaffe, and Jin Wang. Control-Variate Estimation Using Estimated Control Means. *IIE Transactions*, 44(5):381–385, 2012.

[52] B. Peherstorfer, K. Willcox, and M. Gunzburger. Survey of Multifidelity Methods in Uncertainty Propagation, Inference, and Optimization. *SIAM Review*, 60(3):550–591, 2018.

[53] Benjamin Peherstorfer. Multifidelity Monte Carlo Estimation with Adaptive Low-Fidelity Models. *SIAM/ASA Journal on Uncertainty Quantification*, 7(2):579–603, 2019.

[54] Benjamin Peherstorfer, Karen Willcox, and Max Gunzburger. Optimal Model Management for Multifidelity Monte Carlo Estimation. *SIAM Journal on Scientific Computing*, 38(5):A3163–A3194, 2016.

[55] Hadi Pouransari and Ali Mani. Effects of Preferential Concentration on Heat Transfer in Particle-Based Solar Receivers. *Journal of Solar Energy Engineering*, 139(2):021008, 2017.

[56] Ernesto E. Prudencio and Karl W. Schulz. The Parallel C++ Statistical Library QUESO: Quantification of Uncertainty for Estimation, Simulation and Optimization. In *Euro-Par 2011: Parallel Processing Workshops*, pages 398–407. Springer, 2012.

[57] M. Rahmani, G. Geraci, G. Iaccarino, and A. Mani. Effects of Particle Polydispersity on Radiative Heat Transfer in Particle-Laden Turbulent Flows. *International Journal of Multiphase Flow*, 104:42–59, 2018.

[58] Pamphile T. Roy, L. Jofre, J. C. Jouhaud, and B. Cuenot. Versatile Adaptive Sampling Algorithm Using Kernel Density Estimation. *European Journal of Operational Research*, 2019 (under review).

[59] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In *Graphics Hardware 2007*, pages 97–106. ACM, August 2007.

[60] Oliver Sinnen. *Task Scheduling for Parallel Systems*, volume 60. John Wiley & Sons, 2007.

[61] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A High-productivity Programming Language for HPC with Logical Regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 81:1–81:12, New York, NY, USA, 2015. ACM.

[62] Elliott Slaughter, Wonchan Lee, Sean Treichler, Wen Zhang, Michael Bauer, Galen Shipman, Patrick McCormick, and Alex Aiken. Control Replication: Compiling Implicit Parallelism to Efficient SPMD with Logical Regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 14. ACM, 2017.

[63] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[64] James R. Stewart and H. Carter Edwards. A Framework Approach for Developing Parallel Adaptive Multiphysics Applications. *Finite Elements in Analysis and Design*, 40(12):1599–1617, 2004.

[65] Shankar Subramaniam. Lagrangian-Eulerian Methods for Multiphase Flows. *Progress in Energy and Combustion Science*, 39:215245, 04 2013.

[66] Geoffrey Ingram Taylor and Albert Edward Green. Mechanism of the Production of Small Eddies from Large Ones. *Proceedings of the Royal Society of London. Series A-Mathematical and Physical Sciences*, 158(895):499–521, 1937.

[67] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM, 2013.

[68] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[69] Hilario C. Torres, Manolis Papadakis, Lluis Jofre, Wonchan Lee, Alex Aiken, and Gianluca Iaccarino. Soleil-X: Turbulence, Particles, and Radiation in the Regent Programming Language. In *Proceedings of the Parallel Applications Workshop, Alternatives To MPI+X*, PAW-ATM '19, 2019.

[70] Mariano Vázquez, Guillaume Houzeaux, Seid Koric, Antoni Artigues, Jazmin Aguado-Sierra, Ruth Arís, Daniel Mira, Hadrien Calmet, Fernando Cucchietti, Herbert Owen, et al. Alya: Multiphysics Engineering Simulation Toward Exascale. *Journal of computational science*, 14:15–27, 2016.

[71] David W. Walker and Jack J. Dongarra. MPI: A Standard Message Passing Interface. *Supercomputer*, 12:56–68, 1996.

[72] Xiaohu Wu, Patrick Loiseau, and Esa Hyytia. Efficient Algorithms for Scheduling Moldable Tasks. *arXiv preprint arXiv:1609.08588*, 2016.