Production and Research Development Tools for Current and Future Platforms

Presenter: Martin Schulz, LLNL

LLNL has a long history in developing a wide range of debugging, code correctness and performance analysis and optimization tools. Many of these tools explicitly target scalable systems and applications and are being developed for LLNL's exascale efforts. This includes production tools like STAT, Open|SpeedShop, and mpiP as well as active research efforts on race detection (Archer), MPI correctness (MUST) and performance visualization for memory traffic and data transfers (MemAxes), messaging for MPI and task based systems (Ravel) and network contention (Boxfish or DragonView). This talk will provide an overview of these efforts and provide the necessary pointers to these tools that will allow PSAAP centers to deploy them for their debugging and optimization needs.

Gladius: A Framework for Online Distributed Heap Analysis

Sam Gutierrez, LANL

A performance bug is a flaw in a computer program that results in degraded resource utilization of processors, memories, network, or storage. Slow and inefficient program execution due to these sorts of bugs is problematic in all computing environments, but is especially so in high-performance computing environments where these effects are multiplied by system scale. We postulate that the difficulties associated with the identification, diagnosis, and repair of performance bugs are exacerbated in parallel declarative systems, an important class of programming systems slated for emerging heterogeneous supercomputer architectures, because developers are further removed from low-level execution details that ultimately impact an application's overall efficiency, e.g. data representation, movement and placement. When compared with today's more explicit imperative parallel programming environments, e.g. C interfacing to a message passing library, parallel declarative systems tend to offload more responsibility onto an underlying runtime system and therefore tend to obfuscate causal relationships between performance bugs and their origins.

While many of today's parallel programming productivity efforts are focusing on the design and implementation of parallel declarative models and systems, few efforts have focused on providing accompanying infrastructure tailored specifically for these emerging paradigms. Moreover, even though there is a clear need for approaches, mechanisms, and tools to efficiently observe the behavior of parallel declarative programming systems executing across a set of distributed heterogeneous resources, few exist today. In this work we wish to address this need by introducing an approach to support efficient in situ memory analysis of parallel and distributed applications written in a particular declarative style, namely functional with explicit regions.

Optimizing Applications for Future Hardware with Byfl

Scott Pakin, LANL

Ideally, applications currently being designed and developed should run fast on future supercomputers. Once a supercomputer is installed, it is straightforward to tweak compiler options or rewrite small, performance-critical routines to optimize applications for it. However, this approach does not scale with application size and complexity as there may not be a single performance hot spot. Furthermore, by the time a supercomputer is installed it is too late to consider fundamental redesign of core computational methods or data structures in a large-scale application.

In this talk we present a novel approach to performance optimization in the form of *hardware-independent* application analysis. The research question is, How can one discuss application performance without talking about pipeline effects, memory hierarchies, branch penalties, or even time? We argue that architectural trends (e.g., flops becoming relatively cheaper but memory accesses becoming relatively more expensive) indicate regions of code that are likely to perform well or poorly on future supercomputers. Our Byfl tool helps users characterize code regions and data structures to identify "red flags" early in the implementation stage, long before a new supercomputer reaches the machine-room floor.

Caliper: Enabling Performance Annotations for Context-Aware Performance Analysis

Presenter: David Boehme, LLNL

Correlating metrics that reflect performance (e.g., execution time) with program context information (e.g., phases, key data structures, application topologies) is key to understanding HPC application performance, but also requires mechanisms for this context information to be represented. In this talk we introduce a novel approach that consists of (1) an API that application and library developers can use to express context information across independent application modules, (2) a flexible data model that can efficiently represent arbitrary performance-related data, and (3) strategies that allow tools to take advantage for the context information for performance analysis. We demonstrate how this composite approach allows us to easily create powerful measurement solutions that facilitate the correlation of performance data across the entire software stack.

Kokkos: Performance portability for next generation HPC

Presenter: Carter Edwards, SNL

The Center for Computing Research (CCR) developed the Kokkos programming model and library to enable our HPC applications to achieve *performance portability* across the diversity of next generation platforms (NGP); i.e., the identical application source code achieves performant thread-level parallelism on multicore CPU, Intel Xeon Phi, NVIDIA GPU, and future architectures. Kokkos' recent release on github.com and formal tutorials at Sandia and Supercomputing'15 conference has facilitated collaborations with DOE labs, universities, vendors, and other organizations. The Kokkos team is also providing leadership for DOE laboratories' participation on the ISO/C++ language standard committee to subsume Kokkos features into a future C++ standard.

Legion: Mapping and Analysis Tools

Legion is a task-based programming model intended for heterogeneous exascale systems. Legion is relatively new; the first paper on Legion was published in 2012 and the first large-scale application was written in 2014. Current development on Legion is taking place at Stanford, NVIDIA and within the DoE.

A key part of the Legion design is that programs do not commit to where computations ("tasks") and data are placed in the machine. The binding of tasks to specific processors and data to specific memories is the role of a separate *mapping.* The purpose of separating mapping from the rest of the program is to enhance performance portability: by not baking in decisions about where tasks and data will live, it becomes much easier to move computation and data to different places in a machine or to move entire programs from one machine to another.

Mappings in Legion are dynamic and under application control. When a task is launched, the Legion runtime executes a number of standard call-backs provided by the application that make the mapping decisions. For example, the first mapping decision is to select a processor on which the task will execute; another mapping decision is to pick a concrete memory that will hold each of the task's arguments (which, in general, are large collections of data). Other parts of the mapping interface determine which variant of tasks should be executed (if multiple variants have been supplied) and work stealing policies, among other things.

The typical approach to developing a Legion application is to first to write something that is functionally correct using a provided default mapper. Once the program is working, it is then tuned primarily by improving the mapping decisions, taking advantage of the programmer's knowledge of what is likely to work best for that particular program. To evaluate and refine mappings, performance understanding tools are needed. Legion has *legion-prof,* which shows the execution of all tasks in a multi-node system and their relationships to each other.

In this short presentation we'll cover the basics of Legion mapping and the use of legion-prof to understand and improve program performance.

Website: legion.stanford.edu
Contact: Alex Aiken, aiken@cs.stanford.edu

Loopy

Loopy is a programming system for array computations that targets CPUs, GPUs, and other, potentially heterogeneous, compute architectures. Is based on the idea that the mathematical intent and the computational minutiae of a computation should be strictly separated. To attain that goal, loopy realizes programs as objects in a host programming language (Python in this concrete case) that can be manipulated from their initial, "clean", mathematical statement into highly device-specific, optimized versions by ways of a broad array of transformations.

It is distinguished from the directive-based model in at least two ways: First, by being embedded in a full host-side programming language, transformation code can be composable and contain further abstractions designed to simplify and ensure consistency as well as adapt dynamically to target machines and available tuning data.Second, by being based on explicitly requested transformations, loopy makes use of the opportunity to modify executed code in non-equivalent ways (such as kernel fusion and data layout changes) that are difficult to realize under the traditional view in which directives are optional for semantics .

Loopy has been demonstrated to be able to match the performance of hand-optimized code in a number of important application areas, for example a "dynamical core" (i.e. a PDE solver) to be used in a next-generation, heterogeneous-architecture weather model funded by the Office of Naval Research. Loopy's array-based programming model lends itself to most use cases occurring in scientific computing, including dense linear algebra, sparse matrix computations, fast multipole methods, unstructured higher-order finite elements, neural networks, image processing, convolutions, and many more.

Typical Use Cases of Loopy

- Design and implementation of high-performance kernels for array-based computations targeting multi-core CPUs with varying SIMD widths, GPUs, and emerging accelerator architectures, such as specifically Intel's Knights Landing hardware.
- Efficient, automated gathering of performance metrics based on symbolic operation counts
- Rapid experimentation with novel tuning strategies enabled by much-shortened implementation times
- Generation of optimized, device-specific code from legacy FORTRAN 77 computational subroutines
- Code generation for domain-specific languages and other user-built abstractions
- Tooling for autotuners and other higher-level performance infrastructure

Specifications at a Glance

- Supported host platforms: Broadly portable (Linux, OS/X, Windows), Python-based
- Code generation targets: "Plain" C, OpenCL, CUDA, Intel ISPC
- Input languages: Loopy-lang (a flat list of assignments and a loop polyhedron) or FORTRAN 77
- Self-contained, single-file binary distribution available
- (Optional) runtime integration available for OCCA and PyOpenCL

License

- MIT License

Web pages

- http://documen.tician.de/loopy/
- https://github.com/inducer/loopy

Contact

- Andreas Kloeckner (andreask@illinois.edu)

SpatialOps / Nebo

A Domain-Specific Language for Portable Solution of PDEs on Structured Meshses

James C. Sutherland
James.Sutherland@utah.edu
Associate Professor - Chemical Engineering
The University of Utah

Introduction

Application programmers face several challenges in developing high-performance scientific computing applications:

1. Complexity in physical processes being modeled require domain-science expertise.
2. Discretization often requires detailed knowledge of data structures as well as applied mathematics underlying the discretization strategy.
3. Different architectures require different programming languages, etc. to achieve performance.

The variety of architectures currently available, and uncertainty concerning future architectures further complicate this scenario. For these reasons, tools that separate programmer *intent* from *deployment* on a given architecture are needed.

Scope

SpatialOps is a software project that abstracts fields of various types. Nebo is a domain-specific language, embedded in C++, which supports various operations on fields through extensive use of template metaprogramming. Currently, support for operations on structured meshes is provided along with particle-cell interpolants. Discrete interpolant, filter, and derivative operators are supported and the set of operators is easily extended.

Nebo allows operations such as:

```
qx <<= - interpx( tCond ) * gradx( temp );
```

For derivative and interpolant operators defined such that they operate on cell-centered fields and produce face-centered fields, this code may unroll to something like:

```
for( int k=0; i<nz-1; ++k ){
  for( int j=0; j<ny-1; ++j ){
    for( int i=0; i<nx-1; ++i ){
      qx[i][j][k] = - 0.5 * ( tCond[i] + tCond[i+1] ) * ( temp[i+1]-temp[i] ) / dx;
    }
  }
}
```

However, Nebo provides much more power. The same Nebo statement above can also generate code for execution on multicore and GPU architectures. Furthermore, Nebo can nest discrete operators such that:

```
rhs <<= divx( interpx(gamma) * gradx(phi) );
```

can be used to construct the right-hand-side of a PDE: [redacted]. Using this, a full step of Forward-Euler can be written as:

```
phiNew <<= phi + deltat * divx( interpx(gamma) * gradx(phi) );
```

Some other features of SpatialOps & Nebo include:

- Support for masks, which define a subset of the domain over which operations occur.
- Ghost cell support, where fields can have an arbitrary number of ghost cells and the result of a series of (stencil) operations records the number of valid ghost cells.
- Memory pool support on both CPU and GPU.
- Thread-pool support.
- Hybrid CPU/GPU execution support where a field may be resident in multiple locations. SpatialOps provides tools for synchronization and tracking which location(s) have up-to-date versions of the field.
- Synchronous and asynchronous host-device memory transfer facilities.

Examples & Tutorials

Documentation, examples and tutorials (all work-in-progress) are available at https://software.crsim.utah.edu/jenkins/job/SpatialOps/doxygen/index.html

Requirements & Dependencies

1. A modern c++ compiler
2. The boost library
3. CMake

For more information:

Contact Professor James C. Sutherland (James.Sutherland@utah.edu)

Performance Analysis and HPC System Simulation Tools

Presenter: Arun Rodrigues, SNL

Applications and architectures are becoming increasingly complex. To design and optimize future supercomputers (and applications) requires tools that are parallel, flexible, and multi scale. For the past several years, Sandia and other institutions have been developing a series of simulators and performance tools that address these requirements. This talk will examine the Structural Simulation Toolkit (SST) and its use in emerging memory, network, and cache architectures.

Analyzing HPC Interconnects using Network Simulators

Presenter: Abhinav Bhatele, LLNL

The disproportionate increase in the computing capacity of a multi-core node as compared to the injection and network bandwidth on high performance computing (HPC) interconnects can make the network a performance bottleneck. Purchasing additional bandwidth to balance the compute-to-bandwidth ratio can be very expensive. Analyzing communication on current and future HPC interconnects and comparing different network topologies with respect to different metrics such as congestion, performance and dollar costs can help us in understanding networks better and making the right procurement decisions. In this talk, I will present two network simulators that we have developed for analyzing traffic and predicting communication performance on HPC interconnects. Damselfly is a functional model based simulator and TraceR is built on top of ROSS, a parallel discrete event simulation (PDES) engine. I will also present studies we have conducted using these simulators.

Tools for Monitoring Entire HPC Centers: the Sonar Project at LLNL
Presenter: Todd Gamblin, LLNL
Increasingly, performance variability is an obstacle to understanding the throughput of large-scale supercomputers. Two runs of the same code, on the same system, may yield vastly different runtimes, depending on compiler flags, system noise, dynamic scheduling, and shared resources such as memory, filesystems and networks. Understanding an application's performance characteristics requires an increasingly larger number of trial runs and measurements. Analyzing performance measurements from such runs is a data-intensive task.

To address these issues, Livermore Computing is deploying Sonar, a "big data" cluster that will store and analyze performance data from the entire HPC center. Sonar aggregates measurements from the network fabric, filesystem nodes, cluster nodes, applications. It will serve as a central data warehouse for measurements collected by tools. We will give an overview of the Sonar cluster and the tools we have integrated with it. We will also discuss some early analysis done using data from this system.

Charm++ & Adaptive MPI

Charm++ is an object-oriented parallel programming system for high performance computing built on top of an adaptive runtime system. Charm++ encourages over-decomposition of a user program into parallel objects that communicate asynchronously and are scheduled based on message delivery by the runtime system. All parallel entities are easily made migratable (automatically so in AMPI) and the runtime system provides support for scheduling, dynamic load balancing, and fault tolerance.

Adaptive MPI (AMPI) is an implementation of the MPI standard written on top of Charm++, giving MPI application developers access to Charm++'s adaptive runtime system. AMPI allows users to take pre-existing MPI applications written in C, C++, or Fortran, compile with the AMPI compiler wrappers, and then run with application-independent, high-level features such as over-decomposition, dynamic load balancing, and online fault tolerance. It does so by encapsulating MPI ranks into light-weight, migratable user-level threads that are bound to Charm++ objects.

Charm++ and AMPI have been demonstrated to scale up to leadership class systems on a variety of HPC applications. Codes ranging from molecular dynamics to cosmology, epidemiology, and more have been written on top of Charm++ and help drive its development. The PSAAPII center at UIUC, XPACC, has its main simulation code, PlasComCM, running on AMPI, and many mini apps and proxy apps have also been ported with either zero or trivial changes to run on AMPI.

Typical Use Cases of Charm++ & AMPI

- Writing asynchronous, dynamic applications from scratch (Charm++)
- Porting codes from MPI to Charm++ one module at a time (MPI + Charm++)
- Running preexisting MPI applications on an adaptive runtime system (AMPI)
- Running applications on systems with hardware variability and failures
- Easy, tunable use of application-independent features: over-decomposition, message-driven execution, dynamic load balancing, power-aware optimizations, and online fault tolerance
- Parallel composition of multiple modules

Platform Support

- Operating Systems: Linux, Mac, Windows
- Networks: Net (UDP/IP), PAMI (IBM), uGNI (Cray), Infiniband, MPI
- Compilers: GCC, IBM, Intel, Cray, PGI, Clang
- Architectures: x86, POWER, ARM

License

- Charm++/Converse License (source code is freely available)

Webpage

- https://charm.cs.illinois.edu/
- http://charmplusplus.org/

Contact

- Laxmikant (Sanjay) Kale: kale@illinois.edu

DHARMA: Distributed asyncHronous Adaptive and Resilient Models for Applications

Presenter: Jeremy Wilke, SNL

ASC has identified several key software development challenges on the road to exascale. While the several-orders-of-magnitude increase in parallelism is the most commonly cited of those, hurdles also include significantly shortened mean times to interrupt, increased imbalance between computational capacity and I/O capabilities, silent errors, and complex hardware architectures. Asynchronous, task-parallel programming models show great promise in addressing these issues, but are not yet fully understood nor developed sufficiently for the ASC application codes. Detailed insight is required into the level of porting effort, performance, scalability, and fault tolerance characteristics of these approaches. The core mission of the DHARMA (Distributed asyncHronous Adaptive Resilient Management of Applications) project is to assess and address the fundamental challenges imposed by the need for performant, portable, scalable, fault-tolerant programming models at extreme-scale, and ultimately to develop this capability for ASC application codes for ATSx systems.

Moya: A JIT compiler for HPC

Tarun Prabhu, William Gropp

University of Illinois, Urbana-Champaign

In order to extract maximum performance from computers, programmers often find it necessary to hand-code optimizations or rewrite code in an "unnatural" way because the compilers cannot or will not carry out the necessary code transformations. Part of the reason for this is that compilers are not always able to infer properties from the code which will allow them to generate the most efficient code for a given architecture. While the programmer can provide this information through compiler directives, these are almost always compiler-dependent and result in code that is neither portable nor easy to maintain. In general, the more information the compiler has about the program, the better the code it can generate. A JIT (Just In Time), or dynamic compiler has access to run-time information that is not available to an AoT (Ahead of Time), or static, compiler. In many cases, the same properties that are difficult or impossible to infer statically are either trivial to determine or are much more easily inferred. The compiler can then more accurately determine the safety and profitability of several optimizations.

Moya is a JIT programmer-guided JIT compiler built with LLVM. The programmer adds annotations (structured comments or compiler directives) to the code telling the compiler what to JIT-compile and when to JIT-compile it. The optimization process is split into two phases — in the first phase, Moya performs a whole-program analysis at compile-time and caches the results. At run-time, Moya uses these results and run-time information such as the input to the application to recompile a part of the code (which the programmer had specified in the annotations) and executes this recompiled (JIT'ted) code.

In this talk, we shall describe Moya, the annotations that are used to control it and present some preliminary results obtained by using Moya on some benchmarks and a full application code.

Scalable Checkpoint / Restart Library (SCR)

Presenter: Kathryn Mohror, LLNL

The consequence of a larger numbers of components in today's and future high performance computing systems is increased failure rates. Today, systems experience failures on the order of hours or days; however, on future exascale systems, we expect failures to occur at higher frequencies. In this talk, we will give an overview of the problem of fault tolerance on high performance computing systems, focusing on current methods employed by the SCR library for mitigating faults. Following this, we will discuss how we expect failure mitigation methods to evolve and perform on future systems, and features that system software can provide to help fault tolerance libraries.

QUO: Accommodating Thread-Level Heterogeneity in Coupled MPI Applications

Sam Gutierrez, LANL

MPI has been the predominant scientific parallel programming system for the last two decades. An enormous amount of parallel and distributed scientific applications have been developed and tuned specifically for this paradigm. Now, hybridizing MPI codes is becoming mainstream and gaining popularity primarily due to the flexibility and performance potential that this hierarchical approach provides. While MPI+X is gaining popularity, it is not yet ubiquitous. Restructuring large, mature code bases to effectively accommodate new parallel programming systems is challenging and takes a significant amount of time and effort. It is often the case that the cost-to-benefit ratio is simply too high to undertake such a task. Furthermore, it is not uncommon that an MPI everywhere version of a scientific code perform as well or better than its MPI+X instantiation. Because of these factors, coupled scientific codes, i.e. applications comprising many libraries, will be built from a set of libraries using a mix of non-uniform threading levels for the foreseeable future.

In this talk, we present QUO (https://github.com/losalamos/libquo), a production-quality library tailored specifically for MPI+(MPI+X) codes that may benefit from evolving process binding policies during their execution, i.e. ones with thread-level heterogeneity. QUO allows for arbitrary process binding policies to be enacted and reverted during the execution of an MPI application, as different computational phases are entered and exited, respectively. We will also outline some of the challenges associated with thread-level heterogeneity in coupled MPI applications.

Qthreads: A library for lightweight threading

Presenter: Stephen Olivier, SNL

Qthreads is a user-level library for lightweight multithreading on the node, accessible directly using a C API or indirectly through a variety of higher-level programming models. Qthreads has been developed to support graph processing and data analytics applications on commodity hardware using runtime-managed massive multithreading and rich synchronization primitives.

HPC Debugging Made Easy: LLNL's Debugging Tools Strategy

Presenter: Dong Ahn, LLNL

With increased architectural, programming-model and application complexities, HPC debugging often demands heroics effort. In this talk, we will present LLNL's advanced debugging tools aimed at easily diagnosing highly challenging and elusive bugs. The Stack Trace Analysis Tool (STAT) is a mature tool specialized to debug those errors that emerge only at extreme scale; Archer can efficiently and accurately identify data races within large OpenMP programs by building on LLVM and ThreadSanitizer; and ReMPI is a scalable record-and-replay tool to help in debugging non-deterministic MPI applications. The talk will include some of the previously intractable real-world problems that these tools helped resolve, highlighting their use cases.

Spack: A Tool for Managing Exascale Software

Presenter: Todd Gamblin, LLNL

The complexity of HPC software is quickly outpacing existing software management tools. HPC applications and tools alike depend on bleeding edge compilers, numerical libraries, runtime systems, and messaging libraries, and the software ecosystem is expected to become even more diverse at exascale. Performance experts and researchers frequently need to experiment with many different builds of the same code in order to get the best performance, but building large software stacks is often a daunting and time consuming task.

In this talk, we introduce Spack (https://github.com/LLNL/spack), a tool used to build, test, and share production applications and tools at LLNL, NERSC, ANL, and other HPC sites. Spack provides a novel, recursive specification syntax that allows parametric builds of packages and their dependency DAGs. It allows arbitrary library and runtime combinations to coexist in the same environment, and its combinatorial versioning system allows developers to experiment with many different variants of a software stack to do performance parameter studies.