Breakout Session Descriptions

High Performance Languages:

After a long era of only one new dominant programming language per decade (FORTRAN in the 60s, C in the 70s, C++ in the 80s, and Java in the 90s), there is a resurgence of new languages. Languages such as Go, Swift, Hack, Rust, Julia, Scala and Dart are only a few years old, but are already used by thousands of programmers. Another promising trend is domain specific languages (DSLs). How will these trends affect high performance computing? Can high performance language(s) solve many of the productivity and performance portability issues burdening high performance computing? How can we make this a reality?

Future synergies in programming systems for data-intensive and compute-intensive science:

There are natural opportunities for synergy among the challenges facing data-intensive and compute-intensive science, and advances in both are necessary for next-generation scientific breakthroughs. Data-intensive science involves the collection, analysis and management of massive volumes of data from experimental/observational facilities as well as large volumes of distributed sensors. Compute-intensive science involves the execution of computational science workflows with large-scale simulations on high-end computing facilities. It is clear that these two classes of scientific endeavors need to work more synergistically, so that (for example) the latest compute-intensive technologies can be employed to enable data analytics on the scales required for data-intensive science, and the latest data science algorithms can be employed to integrate in-situ analyses into compute-intensive workflows. These synergies are already being realized in the commercial world, where high-end computation capabilities are delivered as cloud computing resources to perform machine learning and other forms of predictive analytics on massive volumes of data in a unified ecosystem.

A major obstacle to realizing this kind of synergy in scientific discovery is the current divergence in the programming systems and software stacks used for data-intensive and compute-intensive science. The software ecosystem for data-intensive science is exemplified by the need for near-real-time analysis to support steering of experiments, as well as analysis technologies built on frameworks using programming languages such as Python, R, Java, and Scala. In contrast, the software ecosystem for compute-intensive science is exemplified by programming languages such as C, C++, and FORTRAN, combined with numerical libraries and frameworks built on MPI. The importance of bringing these two ecosystems together in the future is widely recognized, as data-intensive science becomes increasingly compute-intensive, and compute-intensive science becomes increasingly data-intensive. How can we overcome the current divergence in programming systems and software stacks to realize these synergies? What are the key software elements that could help bridge between data-intensive and compute-intensive science in the future?

Programming systems for data centric computing: demands from user facilities, experiments and sensor networks:

Observational and experimental facilities are creating increasingly massive and complex data sets about important chemical, physical and biological processes; many of these experiments are expecting real-time or near real-time responses to control the experiment or steer the observations. In other cases, these observations are compared with the even large output produced by large and increasingly realistic numerical simulations of the same processes. Many of these data analysis tasks also require data from many different sources and formats. How to effectively support these real-time, near-real-time, distributed data management and analysis workloads? What are the new capabilities that allow a closer coupling of experimental, observational and simulation processes that will drive new scientific insights? What programming support (languages, compilers, runtime systems, OS, tools) is needed to create these new capabilities? Which elements of the software stack need to change the most and why?

Programming systems support for post Moore's computing:

Continuing progress of supercomputing beyond the scaling limits of Moore's Law is likely to require a comprehensive re-thinking of technologies, ranging from innovative materials and devices, circuits, system architectures, programming systems, system software, and applications. To be successful these computational paradigms, including both emerging technologies, such as reconfigurable computing, and, radically new technologies, such as probabilistic, quantum, and neuromorphic computing, will require practical programming systems. The session is designed to foster interdisciplinary dialog across the necessary spectrum of stakeholders: applications, algorithms, software, and hardware to discuss opportunities for new programming systems. Motivating workshop questions will include the following. "How do we capture the salient features of these paradigms in the programming system?" "What programming abstractions might insulate applications from these changes?" "What architectural abstractions should be in place to represent the traditional concepts like hierarchical parallelism, multi-tier data locality, and new concepts like variable precision and resource tradeoff directives?" And, finally, "will OpenMP5 work on my quantum computer?"

Ensuring Correctness for Exascale and beyond:

Ensuring the correctness of high performance scientific program has many aspects, which can involve parallelism issues (races or deadlocks), non-reproducible numerical behavior from reordering of operations (e.g., multiple execution orders for reductions), implementation errors (array indexing, bounds errors), or algorithmic problems. Programming systems in the form of languages, analysis, debugging and testing tools may help with some of these problems, but needs to be directed at the problems that arise in scientific computing codes, which involve arrays, floating point calculations, long runtimes, and huge data sets. Other software communities have seen major benefits by focusing on specific bugs, such as array bounds errors that lead to security loopholes. Are there similar problems in the correctness of scientific codes that can lead to verification of important properties, analysis to either prevent or detect some class of errors, or tools that can address particular correctness challenges? The sophisticated (hand or automatic) optimizations can also lead to bugs in scientific codes. How can formal proofs be used throughout code manipulations for a given platform? What transformations/optimizations are amenable to formal proofs and what types of guarantees are possible? Can formal methods handle real codes with million lines of code and multiple languages? Are loose worst-case guarantees more useful than tighter most-cases properties? Can we build advisory tools for guided modification/verification that handle production codes? Do the techniques/tools exist or do they need to be adapted or created from scratch?