**Brandywine Team questions**

Richard Lethin (Reservoir)

It is pretty well established that we will need to move to a new class of communication avoiding linear algebra algorithms to move to exascale.

* What is the nature of X-Stack tools support required for communication avoiding linear algebra (for sparse and dense)?
* Do we need now to throw out existing linear algebra library implementations and rewrite them to be communication avoiding?
* If we are rewriting them, what is a better way to write these libraries?
* What other algorithmic changes are anticipated for exascale and how does the X-Stack need to adapt to support them?  Examples might be mixed precision algorithms, new kinds of scalable FFT, fast multipole.

Andres Marquez (PNNL)

* What are the differentiating characteristics of the different proposed RTS?
* How is XPI going to accommodate these characteristics?
* How do we measure objectively performance & efficiency improvements across the board?
* What HW support (pie in the sky) would be useful?
* How do we ensure that additional levels of improved programmability abstractions do not sacrifice the underlying RTS performance advantages?
* How do we determine the causality of logical and/or performance bugs in an ocean of asynchronous threads?
* How do we migrate/transition legacy codes?

Rishi Khan (ETI)

**Q1: In what ways will it be possible for a runtime software system to increase useful** parallelism**?**

One of the biggest problems that we will face at exascale is finding enough parallelism in strongly scaling problems. Many problems such as molecular dynamics, adaptive Lagrangian-Eulerian PDE solvers, and graph search problems have extremely medium-grained parallelism that is hard to exploit with current runtime environments such as MPI+OpenMP. How can newer runtime systems exploit medium-grained parallelism in problems such as these?

**Q2: How can a future runtime system reduce *latency* or its effects?**

Latency due to resource access and/or contention causes unknown amounts of delay. Latency In blocking runtime calls (such as recv, barrier), read/write to disk, or even cache misses can greatly extend execution time of programs. As systems grow to millions of threads, if these threads have strong synchronization, latency will be a performance killer. Asynchronous calls such as MPI\_IRecv and aio\_read offer a method of posting a request and polling the result as other work is completed. However, these API’s require one to poll while doing other useful work making it difficult to write asynchronous I/O programs. How can future runtime systems handle asynchrony more elegantly?

**Q3: What means are available through runtime techniques to significantly diminish** overhead **or its effects?**

As with any runtime system or OS, there is overhead in making calls to these systems. The overall speed of the application is bounded by the overhead of these calls. This is especially problematic with fine- or medium-grained synchronization where the execution time of a task may be very small. How can runtimes reduce the overhead of their operation?

**Q4: Are there approaches by which runtime system software could mitigate the effects of** contention **for shared resources?**

Application performance suffers when multiple threads contend on shared resources such as memory, network, and disk. How can runtime systems monitor contention and adjust accordingly? How can runtime systems mitigate the effects of the contention including latency and poor resource utilization (such as cache thrashing)?

**Q5: How will runtime systems provide adequate** resilience **for Exascale?**

It is expected that the mean time between interrupt of an exascale system will be on the order of minutes and the major source of error will be soft errors. Resilience has typically been restricted to the domain of the application (e.g. checkpointing) or the hardware (e.g. ECC). Application developers will need more help from the runtime systems and compilers. How can this be achieved?

**Q6: How will runtime systems contribute to significant reduction of** power consumption**?**

On of the largest problems facing practical exascale is impractical power consumption. It is estimated that if we were to use today’s technology, exascale would come at a power cost of about 1000MW. Realistic exascale power consumption cannot exceed 20MW. Most of the energy savings are expected to come from hardware advances, but improvements brought about by silicon process improvements alone will leave us at least a factor of 5 to 10-fold off of target. How can runtime systems interact with operating systems and smart hardware to bridge this gap?

**Q7: Will the use of runtime systems for extreme scale parallel computers require new** programming models**,** languages**, and** compilers**?**

Over the past 40 years, computer programming has been mostly a serial thought process. OpenMP allows one to sprinkle in a few pragmas in otherwise serial code to exploit some parallelism using the fork-join model. MPI threads are serial threads that communicate with messages. Even CUDA and OpenCL are sequential in nature and expose parallelism in a SIMD way where every thread does the same operation to different data sets. As we move to exascale, we face challenges of exploiting more parallelism, hiding latency, mitigating resource contention, providing resilience, and reducing power consumption. Might it be time to look to new programming models, languages, and compilers? If so, what should these environments look like?

**Q8: Can runtime systems simplify** programmability **and/or** debugging**?**

Face it: programming and debugging parallel programs is hard. What can runtime systems do to help make it easier to program and debug? How much help will come from new programming models, languages, and compilers? How do you envision programming and debugging an exascale program?

**Q9: What are the interrelationships between the runtime and operating system software and how are their respective roles different?**

In the past, runtime systems have lived above operating systems and the two rarely cross paths. The runtime system’s job is to meet the needs of the application across processes and nodes and the OS’s job is to meet the needs of all of the applications living on one node. In the past, the OS was needed to multiplex many applications over scarce resources. As we move into an era with plentiful cores and typically one program occupying a majoring of the cores, what is the role of the OS? And what is the role of the runtime system? Should they communicate more closely? If so, how?

**Q10: Are there other roles for runtime system software in future highly scalable computer systems?**