Software and Exascale Computing

Bill Camp
Intel Corporation

Disclaimer: The views expressed herein are solely those of the author as a member of the scientific community and do not claim to represent those of Intel Corporation in any way.

There is really only one software issue facing us in developing a robust exascale computational economy: scalability. Because of scalability concerns, virtually none of today's applications is ready for exa-ops performance. We have increased system-level computing power about a factor of 1000 every decade for several decades now; and we have had to grow systems to do so. Since Moore's Law is increasing device capability at less than half that amount per decade, we have inexorably invested more money in ever larger systems. In 1997, the largest systems in the world achieved terascale performance with fewer than 10,000 processors; and none of them were multi-core. In 2007, the largest systems in the world achieved petascale performance but had more than 10 times as many processors in doing so. We anticipate that exascale systems will have around a million processors and that those processors will be MPPs themselves—having O(1000) cores. Thus an exa-ops system will have around a billion virtual or real cores.

Scalability faces us in numerous disguises:

Scalability of

1. programmability, debug-ability, and optimization
2. interpretability
3. reliability
4. performance
5. the energy cost of software

Programmability, debug-ability, and optimization:

I have little to say about programmability except to note that there is no single magic-bullet solution to this issue. As noted above, for reasons finding their roots in the physics of CMOS semiconductors, any exascale application in the 2018—20 timeframe will involve $O(10^9)$ threads. No human being can program, debug or optimize directly this many threads. At the same time, no new programming paradigms are credible at this point: it looks like we will use a combination of distributed memory methods (gets & puts, message passing, and incoherent global-address space methods) across the ensemble of processors possibly combined with shared memory methods on-processor. High-level languages may allow us to express that parallelism more effectively—or they may continue to just get in the way of successful parallelism. On the positive side of the ledger, I am convinced that for data-parallel applications, we can use the same kind of automation that has proven successful in areas like geometry and meshing: in data parallel applications, create primitives and extend, replicate, map them onto complex graphical representations to cover the domain of interest. In task-parallel applications, we can use self-similar and hierachical approaches familiar from statistical physics: utilize self-organization combined with automated hierarchy of control to manage complex work queues.

Interpretability:

I have even less to say about interpretability. We are already facing a gap between our ability to generate data and our ability to make sense out of it. Just as terascale applications led ultimately to petabytes of data and petascale applications are starting to generate exabytes of data, exascale applications will generate yoddabytes of data. We will struggle to make interpretation of that much data easy or even doable. Visualization is an obvious but less than desirable and incomplete solution. The human visual cortex can deal with about a gigabyte at a time. So, we will have $O(10^{12})$ times as much data as we can visualize effectively in a single image. And that assumes that we find a way to deal with the storage and computing problems implied by such an

approach. Effective interpretation of such data sets will require advances in cognitive software to turn data into information and information into knowledge and knowledge into insight.

Reliability:

This is an area that properly speaking spans the worlds of hardware and software. Until now, we have separated software reliability from hardware reliability. The former has been the domain of software architecture, software engineering, and mathematics; while the latter has been an integral (some would say not integral enough) part of system architecture and design. At the exascale we can no longer afford that separation. Hardware designers are struggling with how to make systems a thousand times more reliable per bit-operation to keep us at the same level we are at in today's best systems. This is compounded by the fact that energy concerns are driving us inevitably to sub-threshold logic. At the same time, the only reason to do exascale computing is to address ever more complex issues. This will require ever more complex software. Software complexity is the number one cause of unreliability in computation today—well exceeding even hardware's worst efforts! So, we can anticipate that without a radical change in how we handle software resiliency and reliability, we are going to be worse off—much worse off than we are today. One idea is that we build a much higher level of local check-pointing capability into our software and hardware systems. For example, using raided non-volatile memory, we could checkpoint state very often by moving copies of needed application state to nearest neighbor nodes in the system several times a minute perhaps several times a second. Since non-volatile memory is only drawing power when it is in use, this would have minimal energy implications. Dynamically, we can pretty effectively protect correctness of state but correctness of logic poses special challenges. State can be protected at about a 10% energy overhead. Logic correctness requires more invasive approaches with some degree of redundancy that could well exceed the 10% overheads that we have learned to tolerate for state—current R&D focuses on residue checking and redundant multi-threading. However, these have significant energy overheads; and, due to the energy issues discussed below, we are going to be more limited than we should like in protecting logic paths. This will require some degree of cooperation between software and hardware—perhaps identifying at compile time certain critical regions which need stronger correctness guarantees. In any case a serious problem that I believe must be overcome is posed by the brittleness of today's algorithms and applications. We are already generating terabytes to petabytes of new state per second. At exascale we will be generating exabytes of state each second; and a single wrong bit can vitiate the entire calculation. For many scientific calculations we should be able to gracefully tolerate amny kinds of bit errors, indeed the loss of many kinds of local resources. For example, in simulating materials, loss of a processor should not cause inherent failure of the simulation. Think of real materials that are full of defects and faults. We know that we will get for most macroscopic and many microscopic properties the same result for quite different distributions of those defects. Why should we not be able to take advantage of that in our simulations?

Performance:

To a large extent, performance is bounded by the product of the effective speed of the local processor and the communications efficiency of the interconnect fabric. The speed of the processor is largely determined by the ability to issue and retire instructions which in turn is governed by pipeline efficiency and memory system overhead, latency, and bandwidth. Normally, we are used to thinking that communications efficiency is dominant at scale; and that probably remains true. However, due to energy concerns, the efficiency of the processor itself bears special watching: we clearly cannot afford the powerful out-of-order cores supporting both prefetch and speculative execution that characterize today's processors.

From a software point of view, scalability is limited by load imbalance, algorithmic serial complexity and parallel efficiency, communications overhead due to the communications hardware, but also overhead due to the communications software architecture and implementation. One should not dismiss the effect of the programming paradigm and its hardware

implementation. If we insist on a cache-coherent shared memory programming environment, we should understand the cost of implementing such an environment in terms of coherency traffic, synchronization overhead, and memory sub-system conflicts.

Load imbalance will arise from vagaries of the applications but also will occur due to loss of self-synchronization caused by the run-time system, the resource manager, and the operating system. Communications overhead must be diminished by aggressive overlap of communications and computation. At 1 billion threads, if we wish to achieve significant parallel efficiency, we need to keep serial fraction and communications overhead extremely small. If we assume that communications overhead is negligible, Amdahl's Law tells us that the serial fraction must be much less than 10^{-9}. For many weak-scaling problems this may well be achievable. To make sure that communications overhead is also negligible, we must have $\alpha . \omega_{co}$ be much less than unity, where α is the ratio of computational speed to communications speed and ω_{co} is the ratio of non-overlapped communications workload in bytes to computational workload in flops. α is determined by the architecture and is limited by cost and especially by physics. ω_{co} is determined by the computational problem, the code architecture and the algorithmic approach. Unfortunately, physics will prevent us from achieving the kind of balance we wish for in α. We are left to compensate for that in software.

William J. Camp, Ph.D.
Chief Supercomputing Architect
Supercomputing Architecture and Planning
Intel Corporation
505 301 5598
william.j.camp@intel.com