

Hors d'oeuvres for exascale programming

Jeff Hammond (e@anl.gov)

Argonne Leadership Computing Facility and
University of Chicago Computation Institute

19 October 2012



Exascale Programming Challenges

- Concurrency (Parallelism)
- Resilience
- Hardware diversity
- Programming Models

I put “hors d’oeuvres” in the title for a reason; this talk is not meant to be completely satisfying.

Concurrency vs. Parallelism

From Sun's *Multithreaded Programming Guide*:

Parallelism A condition that arises when at least two threads are executing simultaneously.

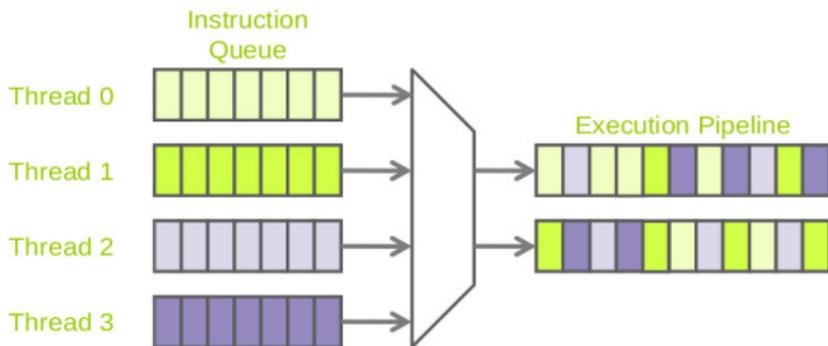
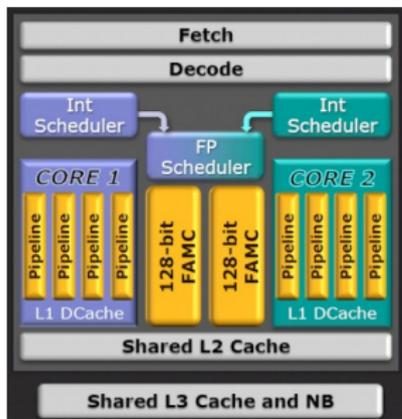
Concurrency A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.

Exascale systems will likely demand billion-way concurrency, but not necessarily because they have billion-way parallelism.

Processing is not going to be the bottleneck, so the aforementioned virtual parallelism may be manifested in the form of simultaneous multithreading (e.g. Blue Gene/Q) and related methods to hide latency. Many threads share execution resources.

Hardware examples

Threads or cores share functional units, so concurrency exceeds parallelism. POWER7, SPARC, Intel Atom all have SMT.



BG/Q has 4 threads sharing ALU+FPU.

See also: NVIDIA memory controller keeps many loads and stores in-flight at once to hide latency; Cray XMT similar concept.

Programming for Concurrency

Traditional languages do not express concurrency effectively:

- FORTRAN 77 relies on compiler but can be autoparallelized.
- C89 and C99 have no memory model; hard to autoparallelize.
- C++98 and C++03 have the same issues as C.
- Fortran 95 FORALL is a very restrictive form of concurrency.

There is progress:

- CUDA is fantastic but obviously not portable. OpenCL?
- OpenMP and OpenACC compensate for serial languages.
- Intel has lots of non-portable bells and whistles.
- Chapel might give us something but will you use it?

Concurrent Algorithms

Life is good if you're a quantum chemist:

$$R_{ij}^{ab} = \sum_{cd} T_{ij}^{cd} V_{cd}^{ab}$$

translates nicely into

```
forall i,j,a,b:  
  forall c,d:  
    R(i,j,a,b) += T(i,j,c,d) * V(a,b,c,d)
```

Not everyone can express their computation in forall loops.

Domain-Independent Languages

Compilers suck because:

- The commodity market is still serial: one app per core.
- Linux, Windows, Office, browsers mostly serial.
- Fine-grain parallelism ignored (except GPU).
- All compiled languages are serial (except CUDA).
- 90:10 rule and low-cost performance monkeys (e.g. games)

HPC is SPMD, desktop is MPMD. If smartphone/tablet are SPMD, that may save us.

Prediction: you'll see a commodity compiler autoparallelize when it extends iPhone battery life by $>20\%$.

Domain-Specific Languages

Take away control from the compiler:

- Specialization allows compiler theory to be practical.
- Restricted semantics required for good autoparallelization.
- Most scientific codes focus on <10 motifs (dwarfs).

Examples:

- Tensor Contraction Engine (TCE): parallelize quantum many-body theory.
- Liszt (Stanford): mesh-based PDEs on multicore and GPUs.
- FENICs/FIAT/Dolfin: FEM PDEs.

Emphasis still on productivity, not performance. SPIRAL is one exception.

Multi-scale (aka Good) Programming

- DSL not need if 90:10 rule applies.
- Humans still smarter than compilers.
- Can always autotune if parameter space reasonable, well-defined.
- Nonorthogonal hardware makes code factorization harder.

Transition from terascale to petascale relied upon scaling MPI of within node or orthogonality of intra- and internode.

Difference between cluster, Cray and Blue Gene is less than 10x in any direction.

Hardware is starting to be >2-dimensional. . .

Resilience and Fault-Tolerance

From Wikipedia:

Resilience is the ability to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation.

Survivability is the ability to remain alive or continue to exist (but potentially at an unacceptable level of service).

Fault-tolerance or graceful degradation is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components.

Application Perspective

Resilience: your application will run fine even when faults are occurring (like the Internet).

Survivability: your application may run poorly when faults are occurring (like my car).

Fault-tolerance: your application will not abort when something fails (like a node), but it might run poorly (because e.g. MPI collectives can't be used).

How many applications running on supercomputers have any of these properties?

MPI isn't the problem. . .

. . . but `MPI_ERRORS_ARE_FATAL` is.

However, if `MPI_ERRORS_RETURN`, this MPI-3.0 function is collective only on the group, not the comm:

```
MPI_COMM_CREATE_GROUP(comm, group, tag, newcomm)
```

If rank i is dead *and you know it*, you can make `MPI_COMM_NEW_WORLD` and proceed.

It's fun to pick on MPI, but other than TCP/IP, can you name a fault-tolerance communication model? UPC, CAF, OpenMP, Chapel, etc. have no discernible plan for fault-tolerance.

Faults aren't even the problem

[T]here are known knowns; there are things we know that we know. There are known unknowns; that is to say there are things that, we now know we don't know. But there are also unknown unknowns – there are things we do not know we don't know.

- Don Rumsfeld



Everyone is scared of hard errors, but soft and silent errors are far more common and a much bigger problem.

Soft and silent errors

Let's imagine that:

- the FPU computes the wrong answer once every 10^{12} instructions.
- Uncorrectable, undetected memory errors (beyond ECC).
- Registers and caches cannot be considered absolutely reliable.

Near-threshold voltage and many other factors will increase this rates substantially from today. $O(10^{15})$ of everything magnifies even femtoscopic error rates.

We can probably count on the vendors to take care of ALU, instruction cache and program counter, but nothing involving FP is mission-critical to OS, Internet, Crypto, banking, etc.

Need for resilience priorities

As part of Andrew Chien's *Global View Resilience* (GVR) X-Stack, we developed the following motifs for resilience in applications (inspired heavily by quantum chemistry).

<u>Application Code Expert</u>		<u>GVR Team</u>
Corrupted Data Property	Response	Priority
Available local	Reread	Low
Available nonlocal	Reread	Low
Available external	Reread	Low
Recomputable w/ local data	Recompute	Low
Recomputable w/ nonlocal	Recompute	Medium
Not Recomputable, Approximable	Approximate	High
Not Recomputable	Restart	Very High

Need for mathematics of errors

Upper bounds on solutions are incredibly powerful:

- If $f(x) \leq g(x) < \epsilon$ and g much less expensive to compute than f , this is not only a recipe for a fast algorithm but also efficient error detection.
- Look for impossible solution data and fix it. Negligible errors are *negligible*, i.e. don't fix what's in the noise.
- Conventional RAS is overkill if you don't need everything to be reliable. Who cares if low-order bits flip?

Know where your error goes:

- Parabolic PDEs (e.g. diffusion eqn.) smooth away errors.
- Hyperbolic PDEs (e.g. wave eqn.) propagate errors until they leave domain.

Hardware diversity

Two swim-lanes:

1) ABLE

Blue Gene/Q + 5 years

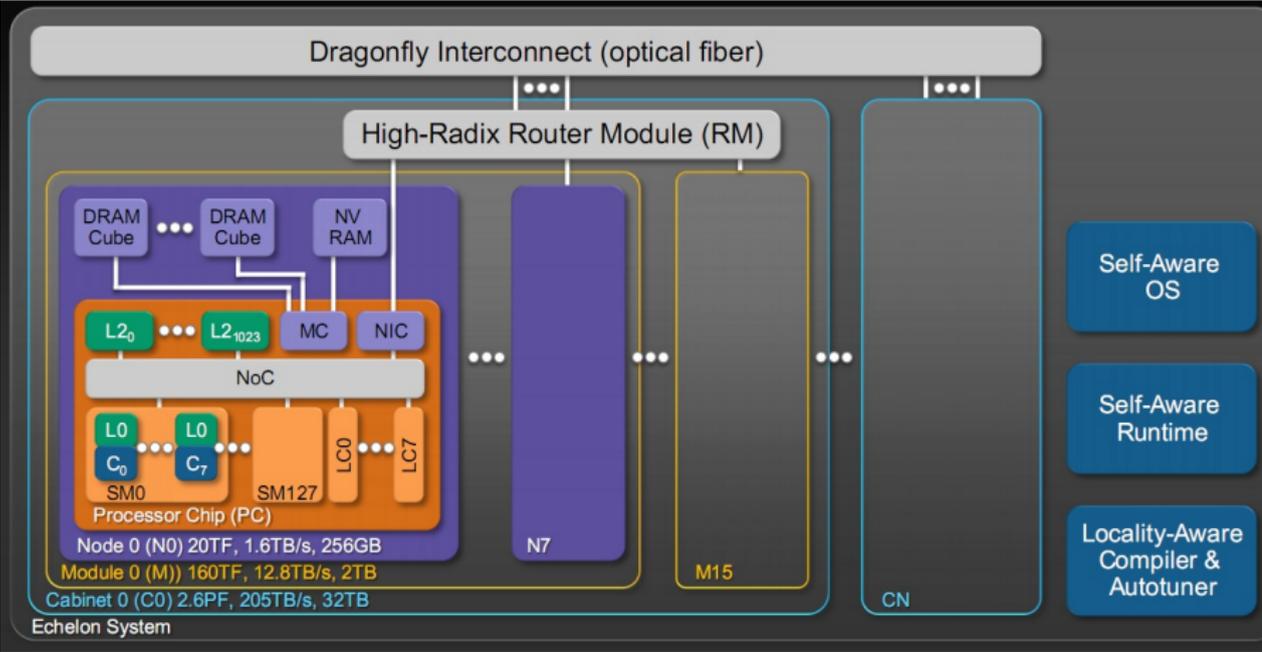
2) SPEC

Cray XK + 5 years



And then Mark Seager, Al Gara, Bob Wisniewski went to Intel and Steve Scott went to NVIDIA while Intel bought Qlogic and Cray's network IP ...

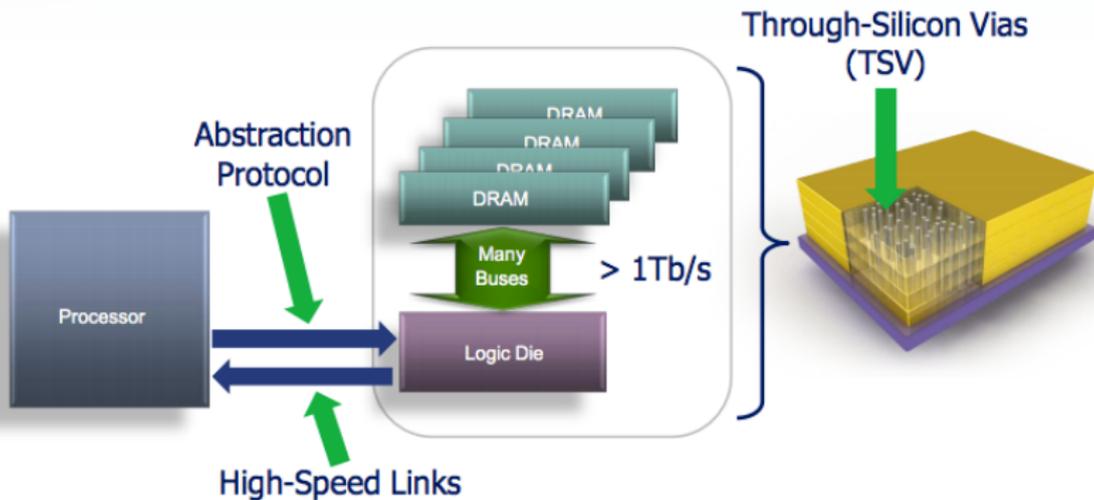
NVIDIA Echelon



From GPU maker to integrator?

Hybrid Memory Cube

Hybrid Memory Cube (HMC)



Notes: Tb/s = Terabits / second
HMC height is exaggerated

Thinking about the future

- The machine won't be 2D ($\text{nodes} \times \text{cores} = \text{MPI} \times \text{Threads}$).
- GPUs rapidly moving back to the die but that doesn't mean the offload model is bad (PIM).
- Logic in/at memory is new and different because we need to think about locality more :-)
- Logic in the memory might be rather "stupid".
- Massive increase in bandwidth won't hide latency.
- How does OS affect memory? Why do we need pages anymore? Do cache lines make sense?

Your guess isn't as good as mine (thanks to NDAs) but I'm still not clear what the right solution is.

Programming Models

MPI is not a programming model except when it is.

- Global Arrays (via ARMCI-MPI), Charm++, UPC/CAF/Chapel (via GASNet), X10 **all** sit on top of MPI. Performance may vary.
- Well-designed numerical libraries such as PETSc, Elemental and PLAPACK have an intrinsic PGAS nature: user pushes data into opaque object store, albeit not with one-sided (yet).
- Research into compiler transformations on MPI codes, esp. turning two-sided into one-sided.
- MPI over OS threads or even user-level threads (e.g. Charm++ AMPI) instead of processes can be much more dynamic.
- MPI-2 dynamic processes are a dynamic model :-)

The MPI-PGAS Wars

All quotes are highly approximate but true in spirit to the best of my ability.

- “We’ve seen from all the NAS Parallel Benchmark studies that people who tune them for UPC go through all same steps as were done for MPI.” – Someone at PGAS12.
- “MPI and UPC are both SPMD, which is a good thing relative to dynamic execution models.” – Kathy Yelick at ICERM.
- “If you want to write fast UPC code, you do explicit communication using `ups_mem{put, get}`” – Lots of people.
- “MPI has performance transparency.” – Rusty Lusk at SC11.

Until there is a PGAS compiler that can optimize data motion, PGAS is going to be just as productive as MPI, which is neither good nor bad. There are no free lunches.

Programming Models and Abstraction

Good programming models raise the level of abstraction:

- Global Arrays is a domain-specific library for dense linear algebra that happens to use one-sided.
- PETSc is a domain-specific library for sparse linear algebra and solvers that happens to use scatter a lot.
- A central concept of Charm++ is over-decomposition and iterative load-balancing. That it has active-messages isn't what makes it wildly successful for some applications.
- UPC is nice in that if performance doesn't matter, you don't have to work as hard at communication because it is implicit.

Every application team needs to decide what is hard about organizing and moving data, then try to find a useful abstraction for this. Why not make it runtime agnostic?

Conclusions

Physics, math, algorithms, software design and performance tuning are all hard.

Keep people focused on what they're good at. Don't let the physicists spend all their time on MPI trivia, etc.

Thanks for your attention.
Dinner will be served in 5-10 years :-)