The Next Quarter-Century at Salishan

Burton Smith
Cray Inc.

The vision for this conference

"…to improve communications, develop collaborations, solve problems of mutual interest, and provide effective leadership in the field of high-speed computing“

How well have we succeeded?

What will our future challenges be?

Communications

We have learned many things at this conference about:

- Monte Carlo
- Functional languages
- Big code development
- Floating point arithmetic
- Lagrangian hydrocodes
- Anthropomorphic programming
- Performance characteristics

We understand each other's language better now.

Collaborations

Salishan has been a fruitful source of collaboration:

- DOE and DOD
  - The historical alliance has remained strong here
- Labs and academia
  - In both computer and computational science
- Academia and industry
  - It’s a great place to meet people
- Labs and industry
  - Including a few “skunk works” projects
- Even labs and labs

<u>Collaboration is a prerequisite of progress</u>

Problem solving

I recall a few small examples:

- One night after dinner, Harry Jordan and I wanted to know "is Gauss-Seidel the same as red-black?"
  - Answers varied from "that can't be true" to "I hope so"
  - Outcome was the 1986 SIAM Journal paper by Harry and Loyce Adams entitled "Is SOR color-Blind?"
- Morven Gentleman and I talked to George Zimmerman about using integers (or unnormalized floats) to make parallel Monte Carlo accumulation reproducible
  - Integer accumulators showed up in George's 1986 talk
- We have also addressed big problems, like the small size and smaller clout of the supercomputer business
  - But we haven't made such things better, at least so far

Leadership

In my estimation we haven't done so well here

- Supercomputing is in deep trouble
  - Architectures — PC-based clusters dominate
  - Languages — MPI reigns supreme
  - Applications — Industry has few, arguably none
- We know there are alternatives to the status quo
  - but we throw up our hands at changing things
- Our future success is in doubt
  - Our supercomputers are impossible to use well and quite difficult to even use poorly
  - Few think "business as usual" takes us to petaflops
- We really need to do something about the situation

Activities for the next 25 years

- Continue to teach each other what we know
- Continue to collaborate on our common problems
- Work to improve high speed computing systems in:
  - Programmability
  - Breadth of applicability
  - Performance
- Work to improve understanding of the issues by others:
  - Government
  - The press

I’ll get the ball rolling by talking about programmability

What's wrong with MPI?

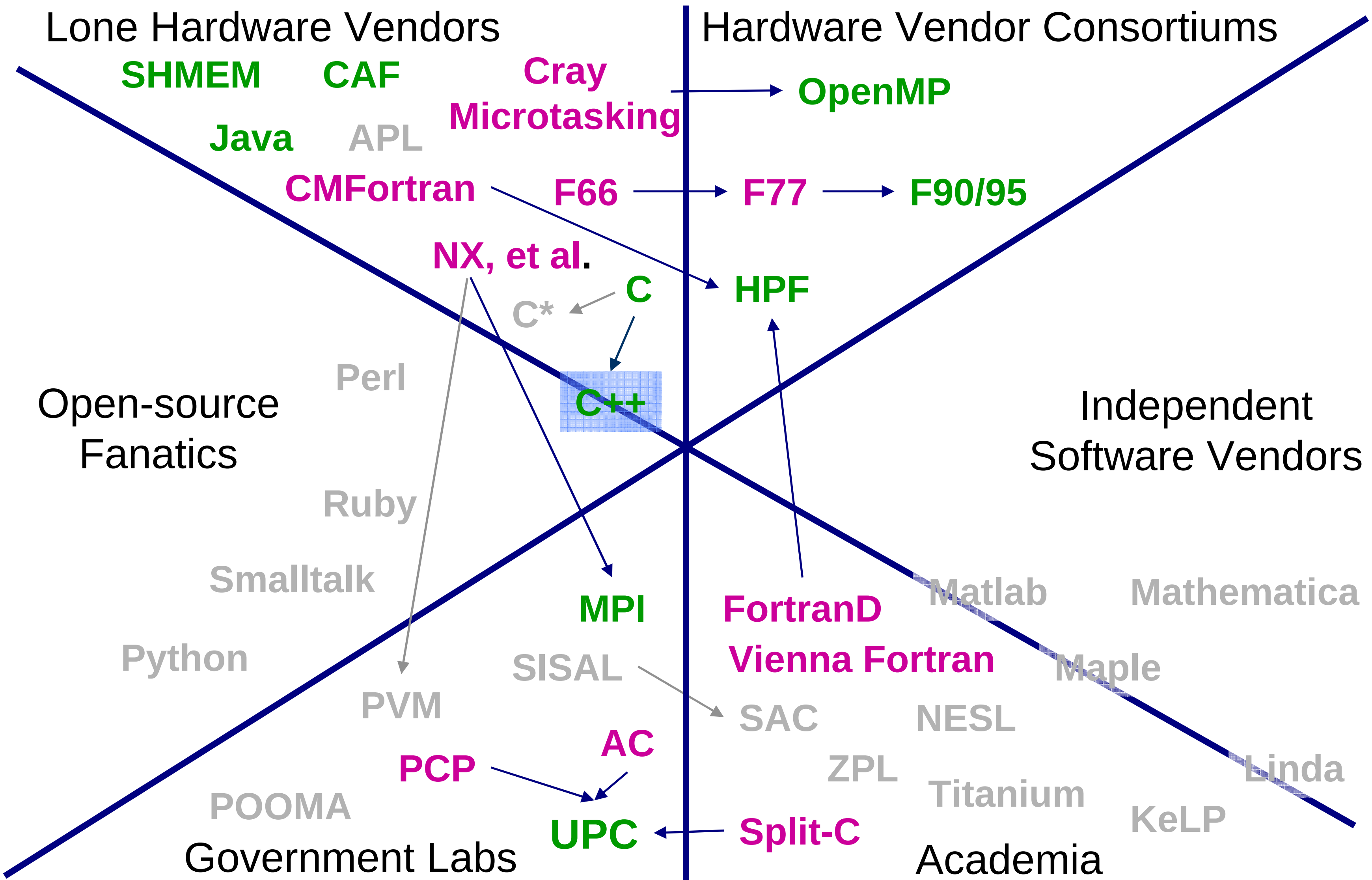
- MPI directly reflects an architectural idea
  - "Nodes" communicating via heavyweight messages
- If MPI is all there is, why build a better architecture?
  - How does the market reward your "additional features"?
- Architecture and language are inextricably linked
  - To improve either, we must improve both
- We need better programming languages:
  - To enhance programmer productivity
  - To allow fine-grain, anonymous communication
  - To enable dynamic scheduling and load balancing
  - To exploit diverse forms of parallelism
  - To improve computer architecture

We used to discuss languages here

- Fortran in all its evolutionary forms
- Functional languages, especially Id and Sisal
- Ada
- SPMD languages like The Force and OpenMP
- Object-oriented languages, especially C++, and Java
- Scripting languages such as Python
- Co-array Fortran, UPC, and Titanium

<u>The “languages” subject has become somewhat moot</u>

Where do languages come from?



Cray is designing a new language

- Its name is *Chapel*
- We have most of the compiler technology needed
  - Dependence analysis
  - Incremental interprocedural fact propagation
  - Loop nest optimizations, *e.g.* wavefronting
  - Dynamic loop nest scheduling
  - Parallelization of general reductions and recurrences
  - Parallelization of memory updates
  - Function inlining
  - Procedure annotations
  - More than just a compiler is required
- We think a new language is necessary to meet the productivity objectives of the DARPA HPCS effort
- We are hopeful it will help us sell computers

Chapel features

- Global view of computation and data structures
- Support for structured data & task parallelism
  - *data:* foralls, domains (dense & sparse arrays, sets, graphs, …)
  - *task:* co-begins, future variables, locale views, …
- Syntactic separation of concerns (locality, parallelism)
- Interprocedurally inferred latent type polymorphism
- Ability to tune for (or ignore) locality using *domains*
- User-extensible distributions, reductions, iterators, …
- Automatic resource management (threads, GC, …)
- Object-oriented features
- Generality
- An open-source implementation

Unsolved language problems

- How to ensure widespread adoption
  - An open-source implementation is necessary, at least
  - It can’t be too awful on typical hardware
- How to reconcile programmability and performance
  - A possible answer is language “telescoping” using interprocedural type inference and cloning
  - This can also help with software re-use
  - Chapel includes this notion
- How to achieve both generality and composability
  - A possible answer is using transactions to preserve invariants on program state
  - Chapel will likely experiment with this idea
  - Maybe a brief discussion of it is called for

The problem with state

- Functional languages tend to be highly composable, but there is no notion of *state* in functional languages
  - Operations on state generally don't commute
- Attempts to add state while preserving commutativity:
  - Applicative State Transition systems (Backus)
  - Monads (Wadler *et al.*)
  - M-structures (Arvind *et al.*)
- A related fact: functional programs are deterministic
  - Introducing state leads to non-determinism (*e.g.* races)
- Some kinds of nondeterminism are good
  - Any ordering that does not affect final results is OK
  - Only the programmer understands the opportunities
  - How can we tell good non-determinism from bad?

A histogramming example

```
const double in[];     //data to be histogrammed
const int f(double);   //f(x) is the bin of x
int h[];               //histogram, initially 0
for(i = 0; i < n; i++)
{   /* (∀int κ)(h[κ] = #{j|0≤j<i ∧ f(in[j])=κ}) */
  int k = f(in[i]);
  h[k]++;
}   /* (∀int κ)(h[κ] = #{j|0≤j<n ∧ f(in[j])=κ}) */
```

- Try to do this in parallel with a functional language!

Histogramming in parallel

```
const double in[];     //data to be histogrammed
const int f(double);   //f(x) is the bin of x
int h[];               //histogram, initially 0
forall i in 0..n-1
{  /* (∀int κ)(h[κ] = #{j|j∈Σ ∧ f(in[j])=κ}) */
  int k = f(in[i]);
  lock h[k];
  h[k]++;
  unlock h[k];
} /* (∀int κ)(h[κ] = #{j|0≤j<n ∧ f(in[j])=κ}) */
```

- Σ *is the set of values i processed "so far"*
- The loop instances *commute with respect to the invariant*
- Premature reads of h[ ] get non-deterministic garbage

What do the locks do?

- The locks guarantee the integrity of the invariant
  - They protect whatever makes the invariant temporarily false
- As long as invariants describe all we care about in the computation and forward progress is made, all is well
  - We have non-determinism “beneath the invariants”
  - In the example, the set Σ captures that non-determinism
- Pretty clearly, the locks need to be lightweight
  - Barriers won’t do the job
- Can we automate or at least verify lock insertion?
  - If we had a language for the invariants, maybe so
- A partial step is to let the language handle the locks
  - This is important to deal with deadlock at least
  - Efficiency is another reason

Atomic transactions on objects

```
const double in[];     //data to be histogrammed
const int f(double);   //f(x) is the bin of x
int h[];               //histogram, initially 0
forall i in 0..n-1 do
{
  int k = f(in[i]);
  with h[k] do {
    h[k]++;
  }
}
```

- This abstraction also allows compiler support and even permits implementation mechanisms other than locks

Nested, multi-object transactions

```
node *m;   //a node in an irregular adaptive mesh
/* (∀node μ)(∀node ν)(ν∈(μ->nbr)* ⇔ μ∈(ν->nbr)*) */
with *m do {               //remove *m from the mesh
  for (n = m->nbr, n != NIL, n = n->nbr){
    with *n do {           //remove link from *n to *m
      for (p = n->nbr, p != NIL, ... //etc
    }
  }
}
```

- In a naive implementation, deadlock could be commonplace
- If a sequence deadlocks or fails, preservation of the invariant requires that it be “undone”, reversing its side effects

Conclusions

- It’s been a memorable and rewarding quarter-century
- The problems we now face are more serious than ever
  - and we may even have forgotten a thing or two
- We need to continue to meet the challenges

“There is no limit to what
we can accomplish
provided
you don’t care who gets the credit”

— George A. Michael