How Shall We Program High Performance Computers?

Burton Smith
Cray Inc.

Parallel programming is still hard

- Programming is too tedious
- Architecture changes too often
- Locality optimization competes with load balancing
- Dynamic load re-balancing changes “who’s where”
- Data layout for irregular data structures is painful
- Memory per node is often insufficient
- Data races are far too common
- Debugging tools are primitive
- . . .

Some of these problems should be correctable.

Languages: chickens or eggs?

- Hardware has driven parallel languages for a while
  - Vectors
    - loop programming practice
    - pragmas and directives
  - Multicomputers
    - PVM and MPI
  - Distributed shared memory
    - shmem and MPI-2 single-sided communication
    - co-array Fortran and UPC
  - Grid computing
    - Java and Jini
- Language efforts for shared memory have languished
  - but there are major issues there as well

Languages should drive architecture

- Languages bridge architecture to applications
- A language should outlive any architecture
  - users need the continuity a language provides
- A language should enhance programmer productivity
  - goodness knows more of this is needed
- Architectures are not programming models
  - shared memory is a good example
- Architectural changes can help language performance
  - especially with communication and synchronization

Avoid message passing[1]

- The sender must know too much about the receiver
  - does the thread still exist?
  - where is it?
  - is it running?
  - is it ready for this message?
- Assembling and disassembling messages is expensive
  - especially with a subroutine interface
- MPI-2 “single-sided” messaging is not much better
  - the receiver has to set up a region
  - the sender still has to know too much
- Message passing is okay for client-server applications
  - but it will more likely be DCOM than MPI

---

[1] between threads

Shmem

- Shmem is a distributed memory access library
- It does put and get to multiple memory “images”
- Communication is processor-to-memory rather than processor-to-processor or thread-to-thread
  - an image can easily support multiple threads per image
    - allowing SMP nodes with multiple threads per processor
  - the threads can be nameless
    - greatly facilitating dynamic scheduling
- Synchronization is memory-based
  - or a system-wide barrier
- Most multiprocessor vendors are implementing it
- Unfortunately, shmem is pretty low-level

Co-array Fortran and UPC

- These languages have distributed data built in
- Addresses are two-dimensional: *image* and *offset*
  - images are identical name spaces, one per "node"
  - subscripting is used to specify which image
  - the programmer controls data layout and scheduling
- Programs directly load and store remote data
  - just as for local data
  - data types are handled by the compiler, not the library
  - the subscripting allows arbitrary communication
- These are significantly higher level than MPI
- Shmem is a potential implementation path
  - but of course native implementations are best

Automatic scheduling: HPF

- HPF offers `block` or `cyclic` data distribution
  - independently for each dimension of an array
- Scheduling is via the *owner-computes rule*
  - `s = u*x +w*v` is computed by the owner(s) of `s`
- The parallelism model is generally *flat*
  - all nodes are working on the same loop nest
  - global barrier synchronization is sufficient
- Compilers for HPF have come a long way
- Extensions have been added for layout of irregular data structures
  - experimentation with these features is ongoing

An array-oriented language: ZPL

- This language comes from Larry Snyder's group at the University of Washington
- It has built-in abstractions for the common cases:
  - data layout including mesh boundaries
  - communications patterns, *e.g.* dimension broadcast
  - high level data operators, *e.g.* reduction and scan
- It is fairly general and exceptionally high level
- The best implementation uses C and shmem
  - The compiler optimizes communication quite well

Bulk-synchrony: BSP

- **The BSP idea is basically**
  `repeat{compute; communicate}until done`
- **Synchronization is removed as a concern**
- **Data layout and scheduling are automatic**
- **Reductions and scans are built-in**
- **The parallelism model is flat**
  - **nested parallelism is up to the programmer**
- **Most computing problems can be solved this way**
  - **given enough communications bandwidth**
- **The BSP idea is especially popular in the U.K.**

Nested parallelism: NESL and ADL

- These languages exploit arbitrary data parallelism
  - sparse linear algebra, for example
- `apply-to-all` or `map` describes the parallelism
- Segmented scans and reductions are also available
- Both need hardware or software multithreading
  - to schedule the heterogeneous work in each node
- The data distribution approach varies
  - NESL linearizes the data to one vector and blocks it
    - this strategy sometimes violates the owner-computes rule
  - ADL uses programmer-supplied partition functions
    - these can vary as the computation progresses

Avoid shared memory[2]

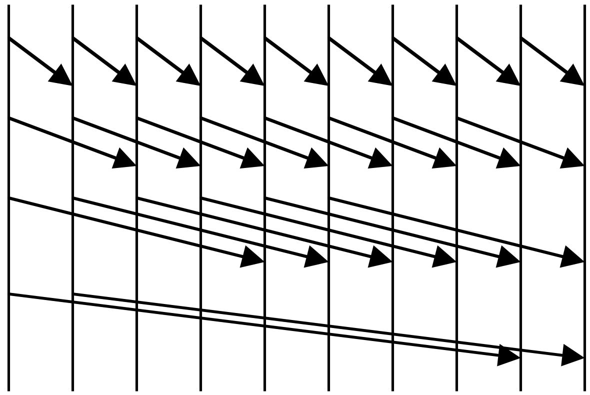
- Variables directly reflect the memory hardware
- Programs schedule values into variables
  - for parallel programs, this is pretty tricky
- Variable references must be properly synchronized
  - barriers
    - tend to oversynchronize the computation
  - wait and signal
    - are better, but need accurate dependence information
- There are alternatives to ordinary variables
  - producer-consumer variables
  - single-assignment variables
  - linear variables

---

[2] as a programming model

Producer-consumer variables

- P-C variables force alternation of loads and stores
  - premature references are forced to wait
- They support *value passing*
  - reductions and recurrences, for example



- They also can implement barriers and wait/signal
- The Cray MTA hardware implements them directly

Single-assignment variables

- These are not variables at all, but dynamic constants
  - any loads that precede the store are forced to wait
- S-A variables can be used to eliminate data races
  - *e.g.* layers of s-a variables instead of barriers
- A key issue with s-a variables is when to reclaim them
  - for efficiency, dependence analysis is required
  - alternatives are reference counts or garbage collection
- The programmer usually knows which load is last

Linear variables

- These are s-a variables that also can only be *used* once
  - there is need for functions that make copies of values
    - Fortran 90 can do this already: `x(k:l,m:n) = y`
  - there is no uncertainty about which load is last
- No reference counting or garbage collection is needed
  - memory management can be very efficient
- Producer-consumer synchronization adds leverage
  - locations can be re-used for sequences of values

Avoid functional programming[3]

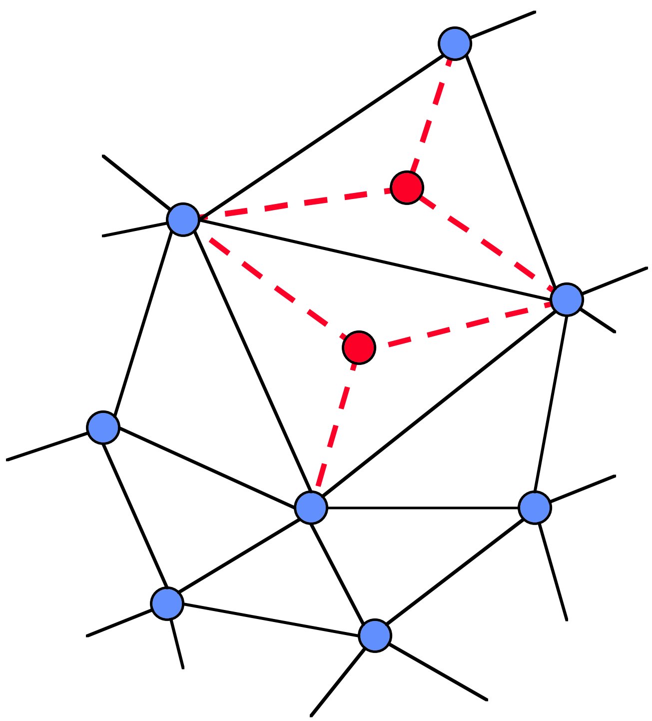
- Dealing with state in functional languages is awkward
  - streams (*e.g.* for I/O)
  - histogramming
  - other updating examples
- Support for "stateful" computation is important
  - for efficiency and expressiveness
- State operations must commute in a generalized sense
  - *i.e.* invariants must be preserved but the final state may differ depending on the order of the operations
- Parallel state transformations are non-deterministic
  - atomicity is required to ensure consistency
- This sounds a lot like data base transactions . . .

---

[3] in its pure form

A transactional example

- **An adaptive irregular mesh needs transactions to create and destroy mesh points safely**



Javaspaces

- **Javaspaces is an adaptation of the Linda language to Java and the Jini distributed object system**
- `lease = space.write(object,txn,lease_req)` **places an** `object` **in the** `space` **as part of transaction** `txn` **for** `lease` **milliseconds**
- `object = space.take(template,txn,timeout)` **takes an** `object` **matching** `template` **from the** `space` **as part of transaction** `txn` **unless** `timeout` **has expired**
- `object = space.read(template,txn,timeout)` **reads an** `object` **matching** `template` **from the** `space` **as part of transaction** `txn` **unless** `timeout` **has expired**
- `txn.commit()` **or** `txn.abort()` **depending on the success of the steps of transaction** `txn`

Transactions on the Cray MTA

- The MTA hardware supports producer–consumer variables using full-empty bits
- A trap occurs after a thread has waited for a while
  - normally, the trap handler then enqueues the thread state for later resumption when the thread can succeed
- Two-phase commit can be implemented by producing incrementally a linked list of the objects to be acquired
  - producing into the link in an object also locks it
  - if the linking process blocks, the trap handler can "back out" by consuming its links in reverse order
- When all objects are locked, the transaction commits
  - object modification must be postponed until then
- Finally, the list is unlocked to complete the transaction

Conclusions

- Parallel programming is still hard
- Languages can help make it easier
  - some may make it harder, so be careful out there
- Language should drive architecture more than it does
  - communication requirements
  - synchronization requirements
- If we want a bigger market for high performance computing, we have to make them easier to use