18.337 Parallel Scientific Computing	March 16, 1995
Lecture 9	
Topic: Parallel Babel	*1995 Scribe: Matteo Frigo*

There notes are based on notes from 1994, by Carmen Perez de la Cruz. This lecture gives an idea of the various options and different moods in the community of parallel scientists, hence the title of the lecture.

1 Some nomenclature

We draw a first distinction between *SIMD* (Single Instruction, Multiple Data) and *MIMD* (Multiple Instruction, Multiple Data). Machines have actually been built using both approaches; examples of SIMD are the CM2 and the MassPar; examples of MIMD are the CM5, the Intel Paragon, the IBM SP. MIMD is more general in the sense that every processor can in principle execute a different program; therefore it can also be a nightmare to program (see below). SIMD is cheaper (for example, the CM2 had a single sequencer sending out instructions to multiple processors); in general the hardware for fetching and decode instructions can be shared among several processors. For this reason SIMD machines usually employ custom processors, while MIMD machines tend to connect off-the-shelf parts together.

The distinction between SIMD and Vector Computer is not so clear and belongs to folklore. The idea is that when you think of SIMD you have in mind many many processors, while Vector Computer usually means a small number (tens) of arithmetic unities driven by a central processor.

The second distinction is between *shared* and *distributed* memory. The former has a central pool of memory shared by all processors (usually on a single bus), and in the latter case each processor has its own private memory; in this case a processors which needs a memory location belonging to another processor must request the memory to it. I remember Tom Leighton saying that the distinction is somewhat weak, since a message sent on a bus to read a memory location is not really different than a message sent over a network to another processor. The idea of shared memory is that no software is involved, and processors are not involved in memory accesses of other processors.

Examples of shared memory: all Crays except T-3D, and also shared memory machines from Sun and SGI. Distributed memory: CM2, CM5, all Intel parallel machines, IBM SP, MassPar, Cray T-3D and T-3E. Beware: KSR fakes shared memory with a software layer. There is no commercially successful large shared memory machine.

As you can imagine, hybrid machines have been built, with some amount of shared memory and some amount of private memory.

2 What is Message Passing?

Message Passing involves the explicit communication between two computing entities where the user specifies when the communicaton occurs as well as the message's data, destination, and identification. Its main feature is that it exposes the cost of communication to the user giving him/her fine control over the communications.

2.1 Nomenclature

- Perfomance:

  Latency: the time it takes to send a zero length message (overhead)
  Bandwidth: the rate at which data can be sent

- Synchrony:

  Synchronous: sending function returns when a matching receiving operation has been initiated at the destination.
  Blocking: sending function returns when the message has been safely copied.
  Non-blocking (asynchronous): sending function returns immediately. Message buffer must not be changed until it is safe to do so

- Miscellaneous:

  Interrupts: if enabled, the arrival of a message interrupts the receiving processor
  Polling: the act of checking the occurrence of an event
  Handlers: code written to execute when an interrrupt occurs
  Critical sections: sections of the code which cannot be interrupted safely
  Scheduling: the process of determining which code to run at a given time
  Priority: a relative statement about the ranking of an object according to some metric

2.2 The Development of Message Passing

- Early Multicomputers:
  UNIVAC
  Goodyear MPP (SIMD.)
  Denelcor HEP (Shared memory)

- The Internet and Berkeley Unix (High latency, low bandwidth)

  Support for communications between computers is part of the operating system (sockets, ports,remote devices)
  Client/server applications appear

- Parallel Machines (lowest latency,high bandwidth - dedicated networks)

  Ncube
  Intel (Hypercubes, Paragon)
  Meiko (CS1,CS2)
  TMC (CM-5)

- Workstation clusters (lower latency, high bandwidth, popular networks)

  IBM (SP1,2) - optional proprietary network

- Software (libraries):

  CrOS, CUBIX, NX, Express
  Vertex
  EUI/MPL (adjusts to the machine operating system)
  CMMD
  PVM
  PARMACS,P4,Chamaleon
  MPI (analgum of everything above and adjusts to the operating system)

- Systems:

  Mach
  Linda (object-based system)

2.3 Machine Characteristics

- Nodes:

  CPU, local memory,perhaps local I/O

- Networks:

  Topology: Hypercube,Mesh,Fat-Tree, other
  Routing: circuit, packet, wormhole, virtual channel random
  Bisection bandwidth (how much data can be sent along the net)
  Reliable delivery
  Flow Control
  "State" : Space sharing, timesharing, stateless

- Network interfaces: Dumb: Fifo, control registers
  Smart: DMA (Direct Memory Access) controllers
  Very Smart: handle protocol managememt

2.4 CM-5 /CMOST Environment:

- Sparc (or sparc superscalar) nodes (optional vector units). 32-128 MB memory
- Sparc (or sparc superscalar) control processor.
- Dedicated Network (Fat-tree). Two data router links per node, 20 MB/s each.
- Separate cooperative communications support in network
- Bisection Bandwidth is 5 MB/s/node (higher on CM-5E)

- Network support for all timesharing (all fall down)
- CMOST operating system:
  Unix (SunOS) derived
  Time sharing
  Data Parallel antecedents
  Single process model
  Memory mapped NI

3 Mechanics of Message Passing :

3.1 Sending and receiving

3.1.1 Sending and receiving:

- CMMD_send_block (destination, tag, buffer, len)
  CMMD_receive_block(source, tag, buffer, len)
- A pairwise synchronizing operation
- High level "Heavyweight" functions
- Designed to prevent network deadlock if used properly ("Safe")
- Relatively high latency:
  Handshaking protocol used
- Bandwidth limited to unidirectional rates
- Resources:
  Messages buffered by sender
  Send side queue: waiting ACK
  Receive side queues: ready to send, waiting to receive
- Interrupts used but not required (not a user concern)
- All packets managed by the system
- Send and receive must match characteristics:
  tag match (or wildcard on receive)
  source match (or wildcard on receive)
  Only min(slen, dlen) bytes actually sent

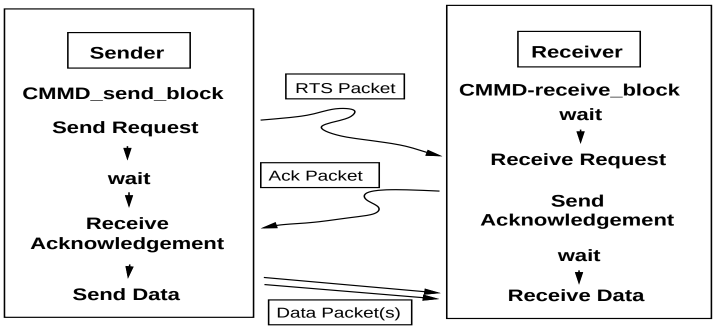

Figure 1: CMMD Cooperative Message Passing Protocol

3.1.2 Receiving considerations:

- Can post receive before send
- Can query the ready to receive queue for specific messages:
  CMMD_msg_pending (src, tag)
- If a desired message is pending, can query for characteristics:
  CMMD_bytes_received()
  CMMD_msg_sender()
  CMMD_msg_tag()
- Can then allocate desired buffer size and post a receive

3.2 Bidirectional transfers:

- CMMD_send_and_receive (source,tag,inbuf,len,dest,tag,outbuf,len)
- Higher latency (setup both in and out transactions)
- Higher bandwidth (don't have to reenter inner kernel twice)
- Allows for more data parallel like communication (cshifts etc.)

3.3 Adding send side buffering:

- CMMD_send_noblock (dest, tag, buf, len)
- Returns immediately (either sends message or copies it into a system buffer)
- Copying data can be expensive.
- Dangerous - system buffer space may not be available.
- Provides behavior compatible with NX csend
- Europeans frequently expect/request this style of messaging.
- Hides latency?
- Same protocol as CMMD_send_block

3.4 'One touch' non blocking messaging

- CMMD_send_async (dest, tag, buffer, length, handler, h-arg)
  CMMD_receive_async(source, tag, buffer, length, handler, h-arg)
- Returns a message Control Block (MCB) immediately
- Contract: no user buffer changes until MCB indicates operation is done
- Handlers are invoked when operation is complete:
  Handler is a function of two arguments: MCB and h-arg.
  Handler is executed with interrupts disabled.
- Handlers are invoked 'asynchronously' with respect to the main user thread of control. Introduces the notion of a critical section.
- Query functions required:
  CMMD_mcb_node(mcb)
  CMMD_mcb_source(mcb)
  CMMD_mcb_tag(mcb)
  CMMD_mcb_bytes(mcb)
- Querying for message arrival:
  CMMD_mcb_pending(source, tag)
- Querying for completion:
  CMMD_msg_done(mcb)
  CMMD_all_msgs_done()

- Waiting for message completion:
  CMMD_msg_wait(mcb)
  CMMD_all_msgs)wait()

- Resources:
  Same queues as non-blocking funcitons
  MCBs are a finite resource: CMMD_mcb_free(mcb);
  Dependence on interrupts.

- Actual transaction is synchronizing, but user code does not wait for synchronization(latency hiding).

3.5 Optimizing for repetitive communications patterns: channels.

- Saves latency: handshake is performed only once. Resources reused.

- Channel state: a slightly different view of communication management.

- Behavior is asynchronous

- Channels are unidirectional.

- Establishing a channel:
  CMMD_open_send_channel(dest, tag, buffer, len).
  CMMD_open_receive_channel(source, tag, buffer, len).
  Sending and receiving functions must agree in tag and buffer length.
  A Channel Descriptor $xcd|x = (r, s)$ is returned by each function.
  This is similar, in constitution and function, to an MCB.

- Operation on a channel:
  CMMD_write_channel(scd)
  Sends a message when the channel is writable.
  Message arrives in receiver's buffer without notice.

- Querying channel state:
  CMMD_send_channel_status(scd)
  CMMD_receive_channel_status(rcd)
  CMMD_is_channel_writable(scd)
  CMMD_is_channel_readable(rcd)

- Allows user to structure sode to avoid latency. Check for message arrival only when algorithmically necessary. Wait only when necessary.

- Closing channels. Freeing resources:
  CMMD_close_send_channel(scd)
  CMMD_close_receive_channel(rcd)

CMMD_all_channels_free()
CMMD_all_channels_free_wait()

- Channel descriptors are a finite resource in the same sense as MCBs. They are freed automatically when both the send and receive descriptors are in the closed state.
- Channels are somewhat lower level, less 'safe' functions. There is more than can go wrong, more state to check. The user must trade off the additional performance and flexibility against the additional responsibility.

3.6 Observations on standard message passing:

- Some sort of synchronization or rendezvous is required.
- Programmer must schedule both ends of each transaction. That is, a send must be matched by a receive somewhere (slightly relaxed in case of a channel).
- Overheads (latencies) are relatively high. This encourages the programmer to view communications as expensive to very expensive operations.
- To counter this expense, latency hiding is attempted. The succes of such strategies varies widely depending on the behavior (sophistication) of the underlying hardware (DMA or co-processor available?).
- Latency hiding affects code structure. This effect has already been noticed by programmers and compiler writes who have had to deal with the ever increasing distance between fast processors and their memories. Operand prefetch and instruction scheduling become at once important and more difficults (many more cycles to hide).
- Proposed alternative? Reduce message latency as much as possible. Get rid of extraneous protocol. Don't preclude the user from gaining performance in the gain of safety.

4 Nontraditional messaging: Active Messages and Ports

- An active message is conceptually similar to a UNIX remote procedure call.

  It is a single packet message.
  The first word contains the address of a function.
  The remaining words contain arguments to which the function is to be applied.
  When the message is sent, the function pointer and arguments are copied from registers to the network interface (no unnecessary memory use).
  Upon receipt, the receiving processor immediately constructs a stack frame with the received arguments and executes the functin at the address specified by the pointer.

- Extremely lightweight (low latency)
- Designed as a compiler target (an assembly language for communication). Allows communication to be better integrated with computation
- Ideas borrowed from dataflow philosophy.
- Work done by von Eicken, Culler et al at UC Berkeley (Threaded Abstract Machine project)

4.1 Active Messages:

4.1.1 Active message variants:

- CMAML_requests(dest, ptr, arg1, arg2, arg3, arg4)
  CMAML_reply(dest, ptr, arg1, arg2, arg3, arg4)
  CMAML_rpc(dest, ptr, arg1, arg2, arg3, arg4)
- Pointer arguments specify destination handlers. Handlers execute in interrupt scope.
- Request sends on left DR channel. Interrupt status of sender is not important.
- Reply can only occur in the handler of a request. It is sent on the right DR channel.
- RPC can only be sent with the interrupts disabled. In its handler, any communication can be initiated.
- On CM-5E, longer packet variants of these will be available.

4.1.2 Interrupt control and polling functions:

- CMAML_enable_interrupts()
  CMAML_disable_interrupts()
  CMAML_poll()
- Many polling variants (with unusual semantics)

4.2 Observations on Active messages:

- Original intent was to provide 'surgical' communications control. Handlers are expected to be short. Get the data out of the network and into the destination as quickly as possible.
- Originally designed to work only with a polling interface (compiler would have complete knowledge of communications schedule).

- Extended (bastardized) by TMC to work in the context of interrupts. Also to be used by mere mortals.
- Speed is all. Safety is nonexistent. User control of interrupts allows for all sorts of potential problems. E.g. router deadlock, critical section violations.
- Extremely flexible. Other protocols can be built on top of these. E.g. CMMD.
- Request/reply protocol was implemented to use the CM-5 network in such a way as to prevent deadlock. Blumofe (MIT) notes that this is overly restrictive.
- Can think of active messages invoking new threads of control upon arrival. When combined with interrupts, this may complicate some code (user must consider safety issues).
- The idea of active messages seems simple, but it's actually very recent.

4.3 Rports

- Active messages are good for moving small amounts of data. What about moving large amounts?
- Desire transport with several properties:

  Latency be kept at a minimum as if with active messages

  Protocol should be kept to a minimum. This allows policy to be moved up to higher levels (simplifies design). As much as possible, the mechanish should not prevent the user from doing anything.
- Solution (also from UC Berkeley)

  An rport is a sized destination for data. Once established by the receiver, data may be deposited at the destination by any sender. Once the destination has received the expected amount of data, the optional handler may be invoked.

4.3.1 Allocation and release:

- CMAML_allocate_rport()
  CMAML_allocate_this_rport(rport_id)
  CMAML_free_rport(rport_id)

4.3.2 Setting rport characteristics:

- CMAML_set_rport_addr(rport_id, *buffer)
  CMAML_set_rport_byte_counter(rport_id, count)
  CMAML_set_rport_handler(rport_id, *handler)
  CMAML_set_rport_handler_arg2(rport_id, arg)
  CMAML_set_rport_total_nbytes(rport_id, size)

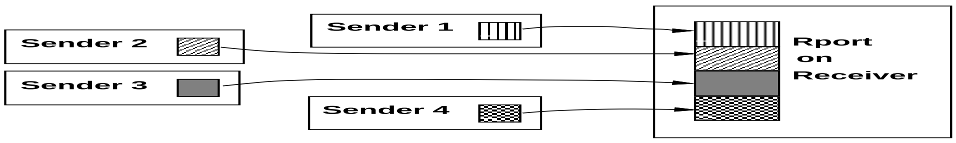

Figure 2: Rport structure

- Total_nbytes is the size of the rport (the amount of data expected)
- Byte_counter is the size less the number of bytes already received
- Query functions exist as well
- Limited number of rports:
  CMAML_number_of_rports()
  CMAML_number_of_free_rports()

4.3.3 Sending to an rport:

- CMAML_scopy(dest, rport_id, dest_align, dest_offset, *buffer, nbytes, *handler, arg).
- Rport_id is the identity of the rport at the destination.
- Dest_align indicates alignment of start of buffer at destination.
- Dest_offset indicates how far into the buffer to start.
- Nbytes is the number of bytes to send.

4.4 Observations on Rports:

- Receiver controls the size of the rport (it can be changed after allocation).
- Receiver must reset byte_counter before rport reuse.
- Senders may execute a handler after sending has been completed.
- Up to the user to create a protocol based on these primitives.

- Many ways to go wrong.
- Very flexible. Can implement many-to-one patterns as well as point-to-point patterns.
- With interrupts enabled, operation is completely asynchronous. No rendezvous or synchronization is required.

4.5 Miscellaneous caveats regarding Active Messages and Rports:

- Remember to poll
- Be careful with interrupts. Be sure to preserve floating point state if necessary:
  CMAML_save_fpu_regs()
  CMAML_restore_fpu_regs()
- Interrupts are expensive.
- Beware of handler recursion. CMMD assigns priorities to message types and handlers:
  Request, Reply
  RPC
  Scopy
  Handlers
  Main thread
- Lower priority items are queued when a higher priority event occurs. The queue is drained (in queue order) before control is return to the main thread. There is limited queue space.
- Flags are volatile. Compilers need to be told that.
- Message order is not preserved (one active message may overtake another).

4.6 Conclusions on message passing

There are dozens of libraries to perform such natural actions as `send` and `receive`. A standard library emerged for message passing (PVM), if you are willing to pay its performance cost. Also, PVM runs on networks of workstations, enabling another form of parallellism; PVM takes care of such issues as different endianity of the processors. Another standard library seems to emerge, called MPI; therefore neither is standard.

5 Programming style

Data-Parallel vs. Message Passing: the explicit communication can be somewhat hidden if one wants to program in a Data-Parallel style; it's something like SIMD: you specify a single action to execute on all processors. Example: is `A`, `B` and `C` are matrices, you can write `C=A+B` and let the compiler do the hard work of accessing remote memory locations, partition the matrix among the processors, etc. Guy Blelloch's doctoral dissertation wants to show that you can take your favourite algorithm and program it in HPF (High Performance Fortran), a data-parallel language with a clever set of primitives.

By contrast, the explicit message passing gives the progammer careful control over the communication, but programming requires a lot of knowledge and is much more difficult (as you probably understood from the previous section).

There are uncountable other alternatives, still academic at this point (in the sense that no commercial machine comes with them bundled). A great deal of research has been conducted on multithreading; the idea is that the programmer expresses the computation graph in some appropriate language and the machine executes the graph, and potentially independent nodes of the graph can be executed in parallel. Example in Cilk (multithreaded C, developed at MIT by prof. Leiserson and his students):

```
thread int fib(int n)
{
    if (n<2)
        return n;
    else {
        cont int x, y;
        x = spawn fib (n-2);
        y = spawn fib (n-1);
        sync;
        return x+y;
    }
}
```

Actually Cilk is a little bit different right now, but this is the way the program will look like when you read these notes. The whole point is that the two computations of `fib(n-1)` and `fib(n-2)` can be executed in parallel. As you might have expected, there are dozens of multithreaded languages (functional, imperative, declarative) and implementation techniques; in some implementations the thread size is a single-instruction long, and special processors execute this kind of programs. Ask Arvind at LCS for details.

Writing a good HPF compiler is difficult and not every manufacturer provides one; actually for some time TMC machines were the only machines available with it. The first HPF compiler for the Intel Paragon dates December 1994.

Why SIMD is wrong: consider the following fragment, where `x` and `y` are vectors:

```
if (x > 0)
     y = x + 2;
else
     y = -x + 5;
```

A SIMD machine will execute both cases, and discard one of the results; it does twice the needed work (see why? there is a single flow of instructions).

Moral: even if the programming model is that there is one processor per data element, the programmer must be aware that it's not true.

Moral II: now that the government doesn't fund this stuff anymore, there is some hope for the future.

Additional comments by Alan Edelman on the past evolution:

CM-2 seemed to be a success ... initiates parallel computing excitement and legitimizes parallel computing as an academic discipline. SIMD worked well with many applications, but was not fully multipurpose.

Masspar decides that the SIMD CM-2 is the way to go and builds machines based on the CM-2 model, but with their own improvements.

Intel has commodity chips available so uses them. Never really came up with the nice software that the CM-2 had. Still can't be programmed in Data Parallel mode? (Rumour has it that a buggy HPF remains in house.)

CM-5 Goes for the commodity chips and gets MIMD for free. Therefore can program either as data parallel or message passing. That the machine was a "fat tree," never seemed to be so relevant. Vector units were slapped on as an afterthought when they realized the machine would not be competitive. Had short term success by beating Intel to market, but the machine became difficult to program for performance. (Message passing did not access vector units.)

In the end, rumour has it that management got into squabbles, and perhaps got greedy, the company fell into bankrupcy protection. Currently morale appears to be so down, and so many people are leaving, it would be hard to imagine a turn-around. Nevertheless a core group of people remain, and though the odds look bad, maybe they can find a way to buck the trend.

Cray half heartedly entered the picture with the T3D and T3E, but I have not heard anything too useful about this machine.

IBM then joined with the SP-2. There is still some hope, but this machine still does not have anything like the software environment of the CM-5.

Silicon Graphics Clusters may be next???

6 Architecture of the CM-5

The machine has 2^n processors (128 in the machine at MIT), each one is in turn a small SIMD machine: there are 4 Vector Units per processors, and a RISC chip to feed them with instructions and data. Each VU achieves about 120 MFLOPS. VU were a later add-on for competitive performance.

The processors are connected by a fat tree, something like a binary tree but whose bandwidth grows as height grows, avoiding therefore the bottleneck at the root. The fat tree looked promising because of good theoretical properties, and they didn't build it as a fat tree anyway (the actual network is a mixture of a fat tree and a butterfly).

The real bottleneck is the interface between the CPU and the network, the actual topology of the network is not very important. Some hacker (Charles Leiserson) suggested to dedicate the CM5 SPARC CPUs to communication, and perform calculations on the vector units.

On the top of the tree sits a front end, a normal sparcstation downloading programs and data to the processors, performing diagnostic checks, and so on.