Porting a Vector Library: A Comparison of MPI, Paris, CMMD and PVM

(or, "I'll never have to port CVL again")

Jonathan C. Hardwick, jch@cs.cmu.edu

http://www.cs.cmu.edu:8001/Web/People/jch

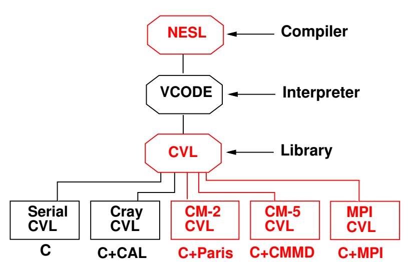
SCandAL group, Carnegie Mellon University

Talk Overview

1. What is the vector library? (CVL)
2. Why was it developed?
3. Experiences porting CVL to Paris, CMMD, PVM, and MPI.
4. Performance of Paris, CMMD and MPI ports.
5. What did we learn? What do we want?

What Is CVL?

- "C Vector Library".
- Specification for a library of vector routines callable from C.
- Back end for high-level parallel languages (NESL, Proteus)



NESL and all CVL implementations are available via FTP.

What Is NESL?

- High-level portable parallel language.
- Used for research on efficient parallel algorithms.
- Tackles the things that Fortran doesn't: dynamic vector lengths, fine-grained communication, nested parallelism.
- Good for unstructured problems with irregular data, e.g. graphs, sparse matrices, meshes, ...
- Current applications: connected components, graph separators, N-body simulations, Delaunay triangulation, ...

Why CVL?

Advantages:

- To port NESL to a new machine, we only need to port CVL.
- Interpretive overhead is amortized over length of vector arguments, which are big for problems we're interested in.
- Use CVL to measure low-level performance of machines.

Disadvantages:

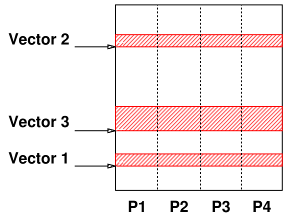
- Single-purpose functions means we lose loop fusion.
- We lose cache locality too. We're interested in unstructured problems that are often cache-unfriendly anyway.

What's In CVL?

CVL functions can be broken down into classes:

- Elementwise: add two vectors, negate a vector, etc.
- Permutation: send some elements of vector to another vector.
- Scans (parallel prefix) and reductions (e.g. global sums).
- Library (sorts, packs, etc): can be built from other functions.
- Glue: memory allocation, timing, scalar-to-vector, etc.

Memory model:



CM-2 CVL: Written in Paris and C

Paris is a *PAR*allel *I*nstruction *S*et for the CM-2.

A set of C functions operating on SIMD bitfields.

Advantages of Paris:

- More flexible than CM Fortran.
- Good match to SIMD architecture.
- Has elementwise, permute, scan and reduce functions.

Disadvantages:

- Not portable. Hard to develop and debug in.
- Not upgraded to use faster "slicewise" f.p. data format.
- Poor match to CVL memory model.

CM-5 CVL: Written in CMMD and C

CMMD is a communication library for the CM-5.

Provides access to message-passing and control networks.

Advantages of CMMD:

- Scans and reductions, using control network.
- Easier to debug, using node-level core dumps.
- Low-overhead message passing using active messages.
  (CM-5 CVL has no buffering. One item to send = one active message.)

Disadvantages:

- Still not portable.
- C wasn't upgraded to use CM-5's vector units.
- CMMD changed constantly.

Why Do I Want Portability?

If I can develop on my workstation, I can avoid:

- Competing for front-end cycles (for compiling and editting).
- Competing for supercomputer cycles (for testing).
- "printf and core-dump" debugging (limited account resources).

If it is portable across supercomputers, I can avoid:

- Porting CVL to a new architecture every 6 months.

PVM CVL: Written in PVM and C

Used PVM 3.0 (could solve some problems using 3.3).

Advantages of PVM:

- Portable to workstations, hence easy to develop and debug.

Disadvantages:

- Low bandwidth (lots of buffering by system).
- High latency.
- No scans or reductions.
- Non-portable if you use manufacturer's extensions to PVM.

MPI CVL: Written in MPI and C

Current using ANL/MSU MPI implementation (MPICH).

Advantages of MPI:

- Portable to workstations, hence easy to develop and debug.
- MPI CVL ported to SP-1, CM-5, Paragon in one day each.
- Different buffering strategies.
- Basic scans and reductions.
- Good support for MPICH (from Bill Gropp and Rusty Lusk).

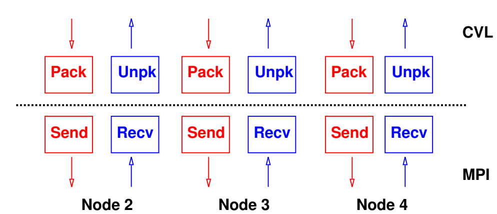
Disadvantages:

- Oriented towards coarse-grained messages (e.g. no active messages on platforms that could support them).

MPI CVL: Implementation

Only interesting implementation issues are in permutations:

- Aggregate messages in buffers before sending.
- Non-blocking send/receive directly from/to these buffers.
- Node 1's buffers on a 4-node machine:



- Given N nodes, each node has up to $4N$ buffers (on a 64 node machine with 4 KB buffers, that's a total of 1 MB per node).
- Global sum to decide when all messages have arrived.

Feature Comparison

	Paris	CMMD	PVM	MPI
Portable	✘	✘	✔✔	✔✔
Easy to debug	✘	✔	✔✔	✔✔
Collective operations	✔✔	✔✔	✘	✔
Efficient communication	✔	✔✔	✘	✔
Good support	✘	✘	✔	✔✔
Stable specification	✔✔	✘	✔	✔✔

PORTING A VECTOR LIBRARY

Results: Communication/Computation (r_∞)

Asymptotic performance of CVL 64-bit floating-point addition vs asymptotic performance of CVL 64-bit random permutation.

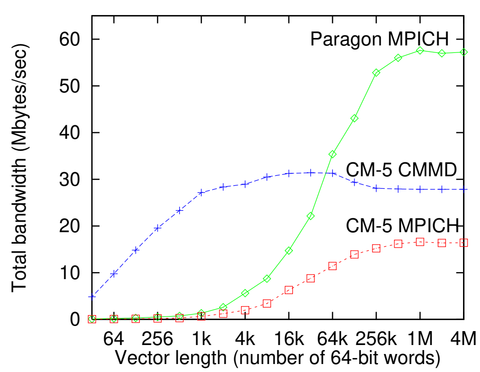
Implementation	Nodes	MFLOPS	Mwords/s	Ratio
SP-1 MPICH	16	117	3.9	30
Paragon MPICH	32	84	7.1	12
CM-5 MPICH	32	32	2.0	16
CM-5 CMMD	32	32	3.5	9
CM-2 Paris	256	87	3.9	22
C90 CAL	1	220	104	2.1

Note: CM-5 implementations do not use vector units. CM-2 figures are for an 8k system (with 256 f.p. chips).

PORTING A VECTOR LIBRARY

Results: Communication Overhead ($n_{\frac{1}{2}}$)

Dependence of all-to-all communication bandwidth on vector length (64-bit random permutation, 32-node systems).



PORTING A VECTOR LIBRARY

Can We Do Better?

Elementwise functions:

- CVL is limited by main-memory bandwidth (but so is e.g. Fortran, on the irregular problems we're interested in).

Because MPI doesn't support fine-grain communication, we have to buffer to amortize message overhead, leading to:

- High startup overhead.
- High per-element overhead. Compare buffering and active message code sequences to send one element on the CM-5. MPI: 6 loads and 3 stores. CMMD: 2 loads and 1 function call.

Scan and reduce functions:

- Collective communication capabilities not yet used by MPICH.

What I Want From Communication Libraries

Having used 4 of them, I've come to the following conclusions:

- If it's not portable to my workstation, I won't use it.
- If it's not portable across supercomputers, I might not use it.
- Don't change the specification (much).
- Provide prompt support.

What I Want From MPI

What I want from an ideal MPI implementation:

- Low fixed overhead.
- Asymptotic performance close to that of the machine.
- Performance numbers for the different primitives.

What I want from MPI-2:

- Support for fine-grained communication (Active messages? Deposit model? Indexed all-to-all sends?)
- Exclusive scans, at least as an option.
- Segmented scans and reductions (HPF already has them...)

Future Work

1. Don't interpret using a fine-grain library; compile everything.
2. Use high-level optimizations to reduce communication.
3. Use different models of data distribution, vector nesting, and control flow made possible by MIMD machines.

We've done (1) and (2) separately in the past.

Now we're combining them and adding (3), with C and MPI as the compiler target.