Seismic Code

• Given echo data, compute under sea map
• Computation model
  – designed for a collection of workstations
  – uses variation of RPC model
  – workers are given an independent trace to compute
    • requires little communication
    • supports load balancing (1,000 traces is typical)
• Performance
  – max mfops = \( O((F * nz * B^*)^{1/2}) \)
  – \( F \) - single processor MFLOPS
  – \( nz \) - linear dimension of input array
  – \( B^* \) - effective communication bandwidth
    • \( B^* = B/(1 + BL/w) \approx B/7 \) for Ethernet (10msec lat., \( w=1400 \))
  – real limit to performance was latency not bandwidth
Database Applications

- **Too much data to fit in memory (or sometimes disk)**
  - data mining applications (K-Mart has a 4-5TB database)
  - imaging applications (NASA has a site with 0.25 petabytes)
    - use a fork lift to load tapes by the pallet

- **Sources of parallelism**
  - within a large transaction
  - among multiple transactions

- **Join operation**
  - form a single table from two tables based on a common field
  - try to split join attribute in disjoint buckets
    - if know data distribution is uniform its easy
    - if not, try hashing
Speedup in Join parallelism

- Books claims a speed up of $1/p^2$ is possible
  - split each relation into $p$ buckets
    - each bucket is a disjoint subset of the joint attribute
  - each processor only has to consider $N/p$ tuples per relation
    - join is $O(n^2)$ so each processor does $O((N/p)^2)$ work
    - so spedup is $O(N^2/p^2)/O(N^2) = O(1/p^2)$

- this is a lie!
  - could split into $1/p$ buckets on one processor
  - time would then be $O(p \times (N/p)^2) = O(N^2/p)$
  - so speedup is $O(N^2/p^2)/O(N^2/p) = O(1/p)$
    - Amdahl's law is not violated
Parallel Search (TSP)

- may appear to be faster than $1/n$
  - but this is not really the case either

- Algorithm
  - compute a path on a processor
    - if our path is shorter than the shortest one, send it to the others.
    - stop searching a path when it is longer than the shortest.
  - before computing next path, check for word of a new min path
  - stop when all paths have been explored.

- Why it appears to be faster than $1/n$ speedup
  - we found the a path that was shorter sooner
  - however, the reason for this is a different search order!
Ensuring a fair speedup

- $T_{\text{serial}} = \text{faster of}$
  - best known serial algorithm
  - simulation of parallel computation
    - use parallel algorithm
    - run all processes on one processor
  - parallel algorithm run on one processor

- If it appears to be super-linear
  - check for memory hierarchy
    - increased cache or real memory may be reason
  - verify order operations is the same in parallel and serial cases
Quantitative Speedup

- Consider master-worker
  - one master and n worker processes
  - communication time increases as a linear function of n
  \[ T_p = T_{COMP} + T_{COMM} \]
  \[ T_{COMP} = \frac{T_s}{P} \]
  \[ \frac{1}{S_p} = \frac{T_p}{T_s} = \frac{1}{P} + \frac{T_{COMM}}{T_s} \]
  \[ T_{COMM} = P \cdot T_{COMM_1} \]
  \[ \frac{1}{S_p} = \frac{1}{P} + \frac{P \cdot T_{COMM_1}}{T_s} = \frac{1}{P} + \frac{P}{r_1} \]
  where \[ r_1 = \frac{T_s}{T_{COMM_1}} \]
  \[ \frac{d(1/S_p)}{dP} = 0 \rightarrow P_{opt} = r_1^{1/2} \text{ and } S_{opt} = 0.5 \ r_1^{1/2} \]

- For hierarchy of masters
  - \[ T_{COMM} = (1+\log P)T_{COMM_1} \]
  - \[ P_{opt} = r_1 \text{ and } S_{opt} = r_1 / (1 + \log r_1) \]
PVM

- Provide a simple, free, portable parallel environment
- Run on everything
  - Parallel Hardware: SMP, MPPs, Vector Machines
  - Network of Workstations: ATM, Ethernet,
    - UNIX machines and PCs running Win*
  - Works on a heterogenous collection of machines
    - handles type conversion as needed
- Provides two things
  - message passing library
    - point-to-point messages
    - synchronization: barriers, reductions
  - OS support
    - process creation (pvm_spawn)
PVM Environment (UNIX)

- One PVMD per machine
  - all processes communicate through pvmd (by default)
- Any number of application processes per node
PVM Message Passing

- **All messages have tags**
  - an integer to identify the message
  - defined by the user

- **Messages are constructed, then sent**
  - `pvm_pk{int,char,float}(*var, count, stride)`
  - `pvm_unpk{int,char,float}` to unpack

- **All processes are named based on task ids (tids)**
  - local/remote processes are the same

- **Primary message passing functions**
  - `pvm_send(tid, tag)`
  - `pvm_recv(tid, tag)`
PVM Process Control

- **Creating a process**
  - `pvm_spawn(task, argv, flag, where, ntask, tids)`
  - `flag` and `where` provide control of where tasks are started
  - `ntask` controls how many copies are started
  - Program must be installed on target machine

- **Ending a task**
  - `pvm_exit`
  - Does not exit the process, just the PVM machine

- **Info functions**
  - `pvm_mytid()` - get the process task id
PVM Group Operations

- **Group is the unit of communication**
  - a collection of one or more processes
  - processes join group with `pvm_joingroup("<group name>")`
  - each process in the group has a unique id
    - `pvm_gettid("<group name>")`

- **Barrier**
  - can involve a subset of the processes in the group
  - `pvm_barrier("<group name>", count)`

- **Reduction Operations**
  - `pvm_reduce(void (*func)(), void *data, int count, int datatype, int msgtag, char *group, int rootinst)`
    - result is returned to rootinst node
    - does not block
  - pre-defined funcs: PvmMin, PvmMax, PvmSum, PvmProduct
PVM Performance Issues

- Messages have to go through PVMD
  - can use direct route option to prevent this problem
- Packing messages
  - semantics imply a copy
  - extra function call to pack messages
- Heterogeneous Support
  - information is sent in machine independent format
  - has a short circuit option for known homogenous comm.
    • passes data in native format then
int main(int argc, char **argv) {
    int myGroupNum;
    int friendTid;
    int mytid;
    int tids[2];
    int message[MESSAGESIZE];
    int c,i,okSpawn;

    /* Initialize process and spawn if necessary */
    myGroupNum=pvm_joingroup("ping-pong");
    mytid=pvm_mytid();
    if (myGroupNum==0) { /* I am the first process */
        pvm_catchout(stdout);
        okSpawn=pvm_spawn(MYNAME,argv,0,"",1,&friendTid);
        if (okSpawn!=1) {
            printf("Can't spawn a copy of myself!\n");
            pvm_exit();
            exit(1);
        }
        tids[0]=mytid;
        tids[1]=friendTid;
    } else { /*I am the second process */
        friendTid=pvm_parent();
        tids[0]=friendTid;
        tids[1]=mytid;
    }
    pvm_barrier("ping-pong",2);

    /* Main Loop Body */
    if (myGroupNum==0) {
        /* Initialize the message */
        for (i=0 ; i<MESSAGESIZE ; i++) {
            message[i]='1';
        }

        /* Now start passing the message back and forth */
        for (i=0 ; i<ITERATIONS ; i++) {
            pvm_initsend(PvmDataDefault);
            pvm_pkint(message,MESSAGESIZE,1);
            pvm_send(tid,msgid);
            pvm_recv(tid,msgid);
            pvm_upkint(message,MESSAGESIZE,1);
        }
        pvm_exit();
        exit(0);
    }
    else {
        pvm_recv(tid,msgid);
        pvm_upkint(message,MESSAGESIZE,1);
        pvm_initsend(PvmDataDefault);
        pvm_pkint(message,MESSAGESIZE,1);
        pvm_send(tid,msgid);
    }
    pvm_exit();
    exit(0);
}
MPI

- **Goals:**
  - Standardize previous message passing:
    - PVM, P4, NX
  - Support copy free message passing
  - Portable to many platforms
- **Features:**
  - point-to-point messaging
  - group communications
  - profiling interface: every function has a name shifted version
- **Buffering**
  - no guarantee that there are buffers
  - possible that send will block until receive is called
- **Delivery Order**
  - two sends from same process to same dest. will arrive in order
  - no guarantee of fairness between processes on recv.
MPI Communicators

- Provide a named set of processes for communication
- All processes within a communicator can be named
  - numbered from 0…n-1
- Allows libraries to be constructed
  - application creates communicators
  - library uses it
  - prevents problems with posting wildcard receives
    • adds a communicator scope to each receive
- All programs start will MPI_COMM_WORLD
Non-Blocking Functions

- **Two Parts**
  - post the operation
  - wait for results
- **Also includes a poll option**
  - checks if the operation has finished
- **Semantics**
  - must not alter buffer while operation is pending
MPI Misc.

- **MPI Types**
  - All messages are typed
    - base types are pre-defined:
      - int, double, real, {,unsigned} {short, char, long}
    - can construct user defined types
      - includes non-contiguous data types

- **Processor Topologies**
  - Allows construction of Cartesian & arbitrary graphs
  - May allow some systems to run faster

- **What’s not in MPI-1**
  - process creation
  - I/O
  - one sided communication