Introduction

- Sample data for program will be on the web today
- Reading
  - Today OpenMP & HPF
  - Thursday DSM papers
    - one paper is only available from the library

OpenMP

- Support Parallelism for SMPs
  - provide a simple portable model
  - allows both shared and private data
  - provides parallel do loops
- Includes
  - automatic support for fork/join parallelism
  - reduction variables
  - atomic statement
    - one processes executes at a time
  - single statement
    - only one process runs this code (first thread to reach it)

Sample Code

```
program compute_pi
   integer n, i
   double precision w, x, sum, pi, f, a
c function to integrate
   f(a) = 4.d0 / (1.d0 + a*a)
   print *, \021Enter number of intervals: \021
   read *,n
c calculate the interval size
   w = 1.0d0/n
   sum = 0.0d0
!$OMP PARALLEL DO PRIVATE(x), SHARED(w)
!$OMP& REDUCTION(+: sum)
   do i = 1, n
      x = w * (i - 0.5d0)
      sum = sum + f(x)
   enddo
   pi = w * sum
   print *, \021computed pi = \021, pi
   stop
   end
```

HPF Model of Computation

- goal is to generate loosely synchronous program
  - original target was distributed memory machines
- Explicit identification of parallel work
  - forall statement
- Extensions to FORTRAN
  - the forall statement has been added to the language
  - the rest of the HPF features are comments
    - any HPF program can be compiled serially
- Key Feature: Data Distribution
  - how should data be allocated to nodes?
  - critical questions for distributed memory machines
  - turns out to be useful for SMP too since it defines locality

HPF Language Concepts

- Virtual processor
  - an abstraction of a CPU
  - can have one and two dimensional arrays of VPs
  - each VP may map to a physical processor
    - several VP's may map to the same processor
- Template
  - a virtual array (no data)
  - used to describe how real array are aligned with each other
  - templates are distributed onto to virtual processors
- Align directives
  - expresses how data different arrays should be aligned
  - uses affine functions
    - align element I of array A with element I+3 of B

Distribution Options

- BLOCK
  - divide data into N (one per VP) contiguous units
- CYCLIC
  - assign data in round robin fashion to each processor
- BLOCK(n)
  - groups of n units of data are assigned to each processor
  - must be exactly (array size)/n virtual processors
- CYCLIC(n)
  - n units of contiguous data are assigned round robin
  - CYCLIC is the same as CYCLIC(1)

Computation

- Where should the computation be performed?
- Goals:
  - do the computation near the data
    - non-local data requires communication
  - keep it simple
    - HPF compilers are already complex
- Compromise: “owner computes”
  - computation is done on the node that contains the rhs of a statement
  - non-local data for the lhs operands are send the node as needed

Finding the Data to Use

- Easy Case
  - the location of the data is known at compile time
- Challenging case
  - the location of the data is a known (invertable) function of input parameters such as array size
- Difficult Case (irregular computation)
  - data location is a function of data
  - indirect array used to access data A[index[I],j] = ...

Challenging Case

- Each processor can identify its data to send/recv
  - use a pre-processing loop to identify the data to to move

```
for each local element I
     receive_list = global_to_proc(f(I))
     send_list = global_to_proc(f^-1(I))
send data in send_list and receive data in receive_list
for each local rhs element I
     perform the computation
```

Irregular Computation

- Pre-processing step requires data to be sent
  - since we might need to access non-local index arrays
- two possible cases
  - gather a(I) = b(u(I))
    - pre-processing builds a receive list for each processor
    - send list is known based on data layout
  - scatter a(u(I)) = b(I)
    - pre-processing builds a send list for each processor
    - receive list is known based on data layout

Communication Library

- How is it different from pvm?
  - abstraction based on distributed, but global arrays
    - provides some support for index translation
    - pvm has local arrays
  - multicast is in one dimension of a array only
  - shifts and concatenation provided
  - special ops for moving vectors of send/recv lists
    - precomp_read
    - postcomp_write
- Goals
  - written in terms of native message passing
  - tries to provide a single portable abstraction to compile to

Performance Results

- How good are the speedup results?
  - only one application shown
  - speedup is similar to hand tuned message passing program
    - one extra log(n) communication operations slows perf
  - how good is the hand tuned program?
    - speedup is only 6 on 16 processors
- What is figure 4 showing?
  - compares performance on two different machines
  - no explanation
    - is this showing the brand x is better then brand y?
    - does it show that their compiler doesn't work on brand y?
  - lesson: figures should always tell a story
    - don't require the reader to guess the story

Communitivity Analysis:Target Environment

- Shared memory multi-processors
- Object oriented programs
  - C^{++} class methods
  - pointer based graph data structures
- Sources of parallelism
  - method invocation
  - methods may be invoked
    - recursively
    - simple looping constructs (converted to tail recursion)

Analysis

- Determine if two method invocations commute
  - intuitive definition: can be performed in any order
  - a followed by b (a;b) is the same as b then a (b;a)
- Technique
  - symbolic evaluation
    - generate symbolic results of running a;b and b;a
    - like running a method but expressions not data
  - compare two results
    - invar analysis - are the variables the same?
      - Need to know basic commutative ops (e.g. addition)
    - sub-method invocation
      - are multi-sets of different invocations the same

Performance Issues

- Method Size
  - methods should be the “natural” size
  - too small - not enough work for overhead
  - too largew -results in a load imbalance
- Synchronization
  - need to provide mutex over shared data
  - granularity an important parameter
    - too small - lock overhead dominates
    - too large - reduce potential parallelism
  - Compiler can change granularity
    - start with one lock per method invocation
    - user lock “coarsening” to merge locks across invocations

Lock Granularity

- Hard to know correct lock size at compile time

  Solution: use runtime adaptation

- Generate multiple versions of methods
  - each uses a different lock granularity
  - provide a way to switch between version
- Adaptation
  - run one at a time and gather timing data for each one
  - select best one
    - need to make sure samples are representative

Questions About the Technique

- Are the speedups good?
  - 50% is not bad for an automatic tool
- Is the technique general?
  - Has only tried two programs
    - these were the target applications from the start
  - works for recursive graph structures
    - how big is this application domain?
- Will it work and play with other approaches?
  - Can data parallelism be used for part of the code?