# Introduction

- Reading
  - Today Communativity Analysis & OpenMP
  - Thursday HPF paper

# Programming Assignment Notes

- Assume that memory is limited
  - don’t replicate the board on all nodes
- Need to provide load balancing
  - goal is to speed computation
  - must trade off
    - communication costs of load balancing
    - computation costs of making choices
    - benefit of having similar amounts of work for each processor
- Consider “back of the envelop” calculations
  - how fast can pvm move data?
  - what is the update time for local cells?
  - how big does the board need to be to see speedups?

# 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
```

# 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?