Introduction

• Parallel Programming is hard
  – Why? It’s just hard

• What if there are good programming tools?
  – Profilers, Debuggers, Tester, ...
  – Tools specialized parallel programming?

• State-of-the-art tools for parallel programming
  – Intel Parallel Studio
  – CriticalBlue Prism
  – VectorFabrics vfAnalyst
• What problems can Parallel Programming Tools help?
  – Where to parallelize?
    • Which code section do I need to look at?
  – Is this effective?
    • What would be the projected speedup?
    • What would be the optimal parallel machine?
  – How can I parallelize the code?
    • Is the code embarrassingly parallel?
    • Otherwise, must guarantee safe parallelization
    • The root cause: data dependences
    • Parallelizable code should have no dependences
      – Except parallelizable reductions

Focus of the paper
• How can we analyze data dependences?
  – Compilers can do the job, but limitations due to the pointer-to-analysis problem

• An alternative: **Data-Dependence Profiler**
  – **Dynamically** analyze data dependences

• How it works?
  
  ```c
  for (i = 1; i < N; ++i) {
    ... = A[i - 1];
    A[i] = ...;
  }
  ```

  – If there is no dependences, it’s *potentially* embarrassingly parallelizable
  – Otherwise, we report the details of discovered dependences for the parallelization
Problems

- However, current data-dependence profilers
  - Too much memory and time overheads
  - Limits features of parallel programming tools
Overview of Prospector

• Prospector
  – Parallel programming assistant tool based on a dynamic data-dependence profiler
  • Finds potentially parallelizable loops
  • Provides detailed dependence information
  • Exploring hidden parallelism
    – Embarrassingly parallelizable? Yes, Prospector guides how to change the code
    – Otherwise, is there possibility to apply other types of parallelism?

  – Need a scalable and rich dependence profiler
    • Even a large and long application can be profiled with detailed information
Input Program
• Source code
• Binary

Prospector
Instrumentation-Time Analysis:
Extracting loops
Data-Dependence Profiling:
Detailed information

Parallelism Explorer:
Easily parallelizable?
Pipeline parallelism? Other models?
How to avoid dependences?
Hints for code modification ...

Programmers
Feedback from Programmers

Database
Scalable Dependence Profiler

• Memory-Scalable Algorithm
  – Key observation: Find compressible patterns
  – Stride-based Dependence Checking Algorithm
  – (1) Detects strides and compresses them
    \[ 10, 14, 18, \ldots, 30 \Rightarrow [10 + 4*i], 0 \leq i \leq 5 \]
  – (2) Computes data dependences with strides
    • A new algorithm: Dynamic-GCD
  – (3) Effectively handles stride-based structures with non-stride structures
• Time-Scalable Algorithm
  – Key observation
    • Dependence profiling itself can be parallelizable
  – A Hybrid Parallelization Model
    • Pipelined parallelization + Data-level Parallelism
  – Additional algorithms for the stride handling

Address space is divided for each task
An Illustration of Prospector

1: void scan_recognize(startx, starty, endx, endy, stride)
2: {
3:   ...
4:   #pragma omp for private (i,k,m,n)
5:   for (j = starty; j < endy; j += stride)
6:     for (i = startx; i < endx; i += stride){
7:       ...
8:       pass_flag = 0;
9:       match();
10:      if (pass_flag == 1) {
11:         if (set_high[tid][0] == TRUE) {
12:           highx[tid][0] = i, highy[tid][0] = j;
13:           set_high[tid][0] = FALSE;
14:         }  
15:         if (set_high[tid][1] == TRUE) {
16:           ...
17:         }  // End of for-i
18:     }  // End of for-i
19: ...
20: void match()
21: {  
22:   reset_nodes();
23:   while (!matched) {
24:     ...
25:     int match_cnfd = simtest2();
26:     if ((match_cnfd) > rho) {
27:         pass_flag = 1;
28:         if (match_cnfd > highest_confidence[tid][winner]){
29:             highest_confidence[tid][winner] = match_cnfd;
30:             set_high[tid][winner] = TRUE;
31:         }
32:     ...
33:   }
34: }
35: void reset_nodes()
36: {  
37:   for (i=0;i<numf1s;i++) {
38:     f1_layer[tid][i].W = 0.0;
39:     Y[tid][i].y = 0.0;
40: }
Conclusion

• We claim the Dynamic Dependence Profiler
  – Will be a basis tool that assists parallel programming

• We present Prospector
  – Parallelism extraction and parallel programming guiding tool
  – Based on a scalable data-dependence profiler
  – Predict parallelizable loops if no dependences
  – Guides code change to avoid dependences
Future Work

• What about hard-to-parallelize loops?
  – Can Prospector guide parallelization for loops which have true dependences?
    • Ideally, we want to provide hints such as “insert locks or use transactional memory at this point to parallelize the loop”
  – Can Prospector give advice on parallelization methodologies?
    • Use multicores for this loop with OpenMP or TBB, but try to GPGPU or SIMD for that loop