Compiler Optimization Techniques for OpenMP Programs

Shigehisa Satoh, Kazuhiro Kusano and Mitsuhisa Sato
Real World Computing Partnership
1. Introduction

- A compiler-directed software DSM for OpenMP
  - Transparent execution of shared-memory parallel programs on clusters of SMPs.
  - Virtual shared memory is provided by software.
  - The compiler insert coherence control codes into programs.
  - Compiler optimization is crucial to obtaining good performance.

- Compiler optimization for OpenMP programs
  - Optimization for coherence operations,
  - Redundant barrier removal,
  - Privatization of dynamically-allocated objects, etc.
2. Outline

- OpenMP on SMP clusters: Our approach
- Dataflow analysis for OpenMP programs
- Optimization for coherence operations
- Performance evaluation
- Summary and future work
3. OpenMP on SMP Clusters

C/C++

Fortran

Front end

Front end

OpenMP Compiler

Backend Compiler

Runtime Lib.

SDSM Threading

NICAM library

Myrinet

SMP SMP SMP SMP

1 Process / Node

Native Threads
4. A Compiler-directed SDSM

- Provide shared memory image by software
- Fine-grain coherence control
5. Coherence Operations

#pragma omp for shared(a,b)
for (i=0; i<n; i++) a[i] = b[i];

set up loop bounds
for (i=lb; i<ub; i++) {
   _check_before_read(&b[i], sizeof(double));
   _check_before_write(&a[i], sizeof(double));
a[i] = b[i];
   _check_after_write(&a[i], sizeof(double));
}

Copy from the home node if the node has a stale copy.

Write back to the home node. Invalidate other copies.
6. Coherence Optimization

- Optimizations within synchronization interval:
  - Merge multiple checks for contiguous locations.
  - Hoist checks for loop invariant variables out of loops.
  - Unify multiple checks for the same location.

- Relaxed memory consistency permits these optimizations without considering thread interactions.

```c
_check_before_read(&a[0], n*sizeof(double));
for(i=0;i<n;i++){
    _check_before_read(&a[i], sizeof(double));
    s += a[i];
}
```

- Our interest is more aggressive optimizations
7. Example Program

A Laplace equation solver: A stencil code

```c
#pragma omp parallel
{
    int i, j, k;
    double err_local, tmp;
    do {
        #pragma omp for nowait
        for (i=1; i<=n; i++)
            for (j=1; j<=n; j++)
                uu[i][j]=u[i][j];
        err_local=0.0;
        #pragma omp for nowait
        for (i=1; i<=n; i++)
            for (j=1; j<=n; j++)
                u[i][j]=(uu[i-1][j]+uu[i+1][j]+uu[i][j-1]+uu[i][j+1])/4.0;
                tmp=fabs(u[i][j]-uu[i][j]);
        if (tmp>err_local) err_local=tmp;
    } #pragma omp critical
    if (err_local>err) err=err_local;
    #pragma omp barrier
    } while(err>1.0e-5);
}
```

- Parallel loop to save the last approximation
- Parallel loop to compute the next approximation
- Iterates until it is converged
8. Sharing Pattern

- The sharing pattern of array uu (static, 4 threads)

This information enables more aggressive optimizations. How to obtain it?
9. Parallel Dataflow Analysis

- Dataflow analysis for explicit parallel programs.
- **Advantages of OpenMP:**
  - Structured parallelism
    - the compiler can recognize structure of the parallel program
  - Relaxed memory consistency
    - consider thread interactions only at synchronization points
  - No race conditions
- **A parallel dataflow analysis framework**
  - A parallel flow graph - models intra-thread flow of control and data synchronization across threads
  - A lattice of dataflow information
  - A set of transfer functions (flow functions)
    - the semantics of OpenMP directives is considered.
10. Parallel Flow Graph

- **Nodes**
  - Sequential nodes: basic blocks similar to those of serial programs
  - Directive nodes: directives or the entry and exit of constructs
  - Synchronization nodes: directive nodes that implies flush operations.

- **Edges**
  - Control edges: flow of control in a single thread.
  - Synchronization edges: event-ordering constraints between synchronization nodes.

```
s=0.0;
Parallel
For(i=0,n,1)
a[i]=(b[i]-b[i-1])/2.0;
tmp=fabs(a[i]-b[i]);
Critical
if(tmp>s)
s=tmp;
Critical
For
Parallel
return s;
s
```
11. Reaching Definitions

- Reaching definitions analysis for OpenMP
  - Take account of both intra- and inter-thread flow of data

- Dataflow equations
  - \( \text{In}(n) = \frac{\text{Out}(p)}{\text{Pred}(n)} \)
  - \( \text{Out}(n) = \text{Gen}(n) - (\text{In}(n) - \text{Kill}(n)) \)

- These equations are the same as the sequential counterpart, but
  - The semantics of the directives are considered when the Gen and Kill sets are computed.
  - Propagation via synchronization edges represents inter-thread flow of data.
12. Dealing with Arrays

- Reaching definitions analysis is extended to deal with array sections.

- **Dataflow information: (stmt, section, flag) where**
  - stmt: an assignment statement in which an array is modified
  - section: array section(s) modified at the assignment
  - flag: represents whether that assignment is globally synchronized or not

```c
#pragma omp for nowait
for(i=1;i<=n;i++) {
    s1:a[i] = ...;
    ... = a[i];
}
statement;
#pragma omp barrier
```

Reaching definitions

- (s1, a[i], false)
- (s1, a[1:n], false)
- (s1, a[1:n], true)

Race condition may occur if a[1:n] is referred
13. Cross-Loop Dependence

- Cross-loop data dependence analysis for Laplace

Flow dependence is found by reaching definitions analysis.

Cross-loop dependence vector for this def-use pair is (1,0).

i.e., the value assigned by the i-th iteration of the first loop is read by the i+1-th iteration of the second loop.

Inter-thread data dependencies can be computed from inter-iteration data dependencies, depending on the scheduling policy.

```c
#pragma omp parallel
{  int i,j,k;
  double err_local,tmp;
  do {
    #pragma omp for nowait
    for (i=1;i<=n;i++)
      for (j=1;j<=n;j++)
        uu[i][j]=u[i][j];
    err_local=0.0;
    #pragma omp single
    err=0.0;
    // implicit barrier
    #pragma omp for nowait
    for (i=1;i<=n;i++)
      for (j=1;j<=n;j++)
        {  uu[i][j]=uu[i-1][j]+uu[i+1][j]+uu[i][j-1]+uu[i][j+1])/4.0;
          tmp=fabs(u[i][j]-uu[i][j]);
          if (tmp>err_local) err_local=tmp;
        }
    #pragma omp critical
    if (err_local>err) err=err_local;
  } while(err>1.0e-5);
}
```
14. Optimized Code

- Optimized code for the first parallel loop in Laplace
  - No coherence operations for non-shared array elements
  - Explicit remote copy between writers and readers
  - Writer-initiated communication
  - Utilize physical shared memory in SMP nodes

```c
for (i=lb;i<=ub;i++)
    for (j=1;j<=n;j++)
        uu[i][j] = u[i][j];
// update the copy on the previous node
if (_is_first_thread_on_node() && _my_node_no > 0)
    _update(&uu[lb][1], sizeof(double)*n, _my_node_no-1);
// update the copy on the next node
if (_is_last_thread_on_node() && _my_node_no < _n_nodes-1)
    _update(&uu[ub][1],sizeof(double)*n, _my_node_no+1);
_barrier();
```
15. Experiments

- **Platform:** COMPaS I
  - A 200-MHz PentiumPro-based SMP cluster
  - Eight 4-way SMP nodes connected via Myrinet
  - The Solaris 2.5.1 operating system
  - The Solaris threads for intra-node parallelism
  - The NICAM library for communication between nodes
16. Benchmarks

- Two data-parallel programs written in OpenMP C
  - Laplace: Solves Laplace equations by Jacobi method. 2048x2048 grid(128MB), 20 iterations. The shared read-write ratio is $O(N^2) : O(N^2)$.
  - JOR: A Jacobi Over-Relaxation solver 4096 variables(64MB), 11 iterations. The shared read-write ratio is $O(N^2) : O(N)$.

- Measured execution times for three versions:
  - Sequential code
  - w/o PDA: Optimized within synchronization intervals
  - w/ PDA: Optimizations based on PDA are performed

- Hand-translated codes and the runtime library prototype are used.
The number of remote copies for shared-writes is greatly reduced.

The performance is saturated due to serialization at the critical section.
18. Performance of JOR

Simple compiler optimization provides good performance.

Performance improvement for shared-read is small.
19. Summary

- **Parallel dataflow analysis for OpenMP**
  - Structured description and relaxed memory consistency enabled efficient and effective analysis

- **Optimization for the compiler-directed SDSM**
  - Explicit data transfer according to the sharing patterns
  - Remove coherence operations for non-shared data
  - Particularly effective for shared-write intensive programs

- **Future work**
  - The compiler is under development.
  - Performance evaluation for more applications
  - Optimizations for other platforms: SMPs, hardware DSMs, page-based software DSMs