

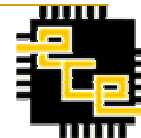
Optimizing Irregular Shared-Memory Applications for Distributed-Memory Systems



Ayon Basumallik and Rudolf Eigenmann

School of Electrical and Computer Engg.
Purdue University

<http://www.ece.purdue.edu/ParaMount>

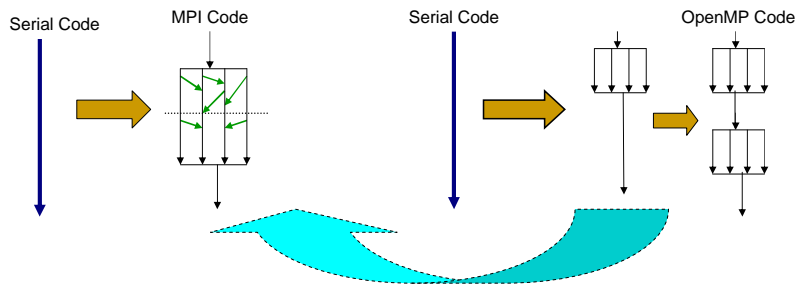


PPoPP 2006

1

Motivation

OpenMP programming has gained popularity and importance for shared-memory platforms, but Message-Passing is still the state-of-art paradigm for distributed-memory platforms.



Extend the ease of OpenMP programming to distributed-memory platforms by **automatic OpenMP to MPI translation**.

2

Motivation

Irregular applications are often difficult to parallelize.

- Converting irregular, serial applications directly to parallel message passing form poses an even greater challenge.
 - Programmer must perform data-to-process mapping even though the actual access patterns are not known till run-time.

```
for(i=0;i<N;i++)  
  A[i] = B[C[i]]+... ;
```

Provide compiler techniques for deploying and optimizing irregular OpenMP applications on distributed-memory platforms

3

Contents

- Brief Introduction to the Baseline OpenMP to MPI Translation Scheme
- Irregular Applications
- Inspection and Loop Restructuring
- Creation of Computation and Communication Threads
- Experiments
- Related Work

4

Baseline OpenMP to MPI Translation Scheme

- Execution Model – SPMD
 - Replicate Serial Regions on all processes
 - Schedule iterations of parallel **for** loops using static block scheduling
 - Allocate shared data on all nodes
- Translation Steps
 1. Identify all shared data
 2. Create annotations for accesses to shared data (use Regular Section Descriptors (RSDs) to summarize array accesses)
 3. Create message sets to communicate data between producers and consumers

5

Baseline Translation

```

/* Loop L1 */
#pragma omp parallel for
for(j=1;j<N;j++) {
    A[j] = foo(x,y,z,i);
}
/* Loop L2 */
#pragma omp parallel for
for(j=1;j<M;j++) {
    B[j] = A[j] + foo2(p,q,r,j);
}
    
```

➤ Static Block Scheduling – on each process compute $lb(1,N,nprocs,rank)$ and $ub(1,N,nprocs,rank)$.

➤ Identify Shared Data – Arrays **A** and **B**

➤ Summarize Accesses

➤ $\langle A, 1, lb_{(rank,1,N)}, ub_{(rank,1,N)}, WRITE_ALL \rangle$

➤ $\langle A, 1, lb_{(rank,1,M)}, ub_{(rank,1,M)}, READ_ALL \rangle$

➤ $\langle B, 1, lb_{(rank,1,M)}, ub_{(rank,1,M)}, WRITE_ALL \rangle$

➤ Compute Message Set

➤ For each pair of processes with ranks p, q :
send from p to q the elements of A in the range

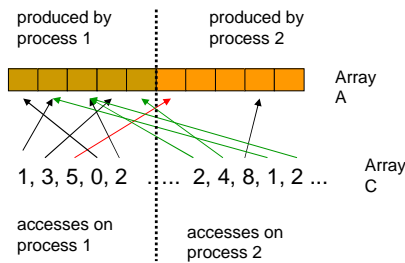
$$[lb_{(p,1,N)}, ub_{(p,1,N)}] \cap [lb_{(q,1,M)}, ub_{(q,1,M)}]$$

6

Irregular Applications

L1 : #pragma omp parallel for
 for(i=0;i<10;i++)
 A[i] = ...

L2 : #pragma omp parallel for
 for(j=0;j<20;j++)
 B[j] = A[C[j]] + ...

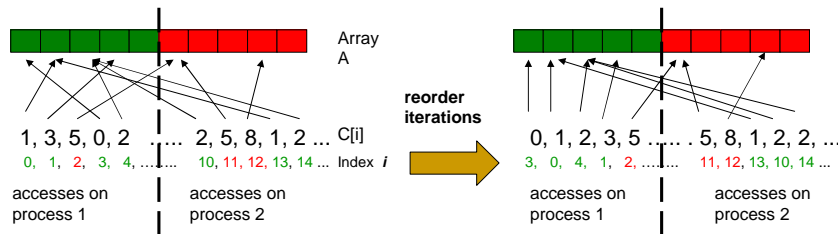


- Subscripts of accesses to shared arrays not always analyzable at compile-time
- Baseline OpenMP to MPI translation –
 - Conservatively estimate that each process accesses the entire array
 - Try to deduce properties such as *monotonicity* for the irregular subscript to refine the estimate
- Still, there may be redundant communication
 - Runtime tests (**inspection**) are needed to resolve accesses

7

Inspection

- Inspection allows accesses to be differentiated (at runtime) as local and non-local accesses.
- **Inspection can also map iterations to accesses.** This mapping can then be used to re-order iterations so that iterations with the same data source are clubbed together.
 - Communication of remote data can be overlapped with the computation of iterations that access local data (or data already received)



8

Loop Restructuring

- Simple iteration reordering may not be sufficient to expose the full set of possibilities for computation-communication overlap.

```
L1 : #pragma omp parallel for
    for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2 : #pragma omp parallel for
    for(j=0;j<N;j++) {
        w[j] = 0 ;
        for(k=rowstr[j];k<rowstr[j+1];k++)
S2:         w[j] = w[j] + a[k]*p[col[k]]
    }
```

Distribute loop L2 to form loops L2-1 and L2-2

```
L1 : #pragma omp parallel for
    for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2-1 : #pragma omp parallel for
    for(j=0;j<N;j++) {
        w[j] = 0 ;
    }

L2-2: #pragma omp parallel for
    for(j=0;j<N;j++) {
        for(k=rowstr[j];k<rowstr[j+1];k++)
S2:         w[j] = w[j] + a[k]*p[col[k]] ;
    }
```

Reordering loop L2 may still not club together accesses from different sources

Loop Restructuring contd.

```
L1 : #pragma omp parallel for
    for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2-1 : #pragma omp parallel for
    for(j=0;j<N;j++) {
        w[j] = 0 ;
    }

L2-2: #pragma omp parallel for
    for(j=0;j<N;j++) {
        for(k=rowstr[j];k<rowstr[j+1];k++)
S2:         w[j] = w[j] + a[k]*p[col[k]] ;
    }
```

Coalesce nested loop L2-2 to form loop L3

```
L1 : #pragma omp parallel for
    for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2-1 : #pragma omp parallel for
    for(j=0;j<N;j++) {
        w[j] = 0 ;
    }

L3: for(i=0;i<num_iter;i++)
    w[T[i].j] = w[T[i].j] +
    a[T[i].k]*p[T[i].col] ;
```

Reorder iterations of loop L3 to achieve computation-communication overlap

The T[i] data structure is created and filled in by the inspector

Final restructured and reordered loop

Creation of Inspector and Executor Loops

- Inspector Loop Creation for a Loop nest L
 - For each shared array A accessed in loop L, allocate an inspection structure
TA : { i_1, \dots, i_n : the index vector of the loop nest L,
 x_1, \dots, x_k : values of all irregular subscripts for the accesses to array A.
}
 - Add statements to L to
 - allocate an instance of TA for every iteration and to assign its elements
 - count the total number of iterations of the loop body of L
- Executor Loop Creation for a Loop nest L
 - Set loop body and loop bounds of executor loop to be identical to the original loop nest L
 - Substitute occurrences of index variables $i_1 \dots i_n$ with $TA.i_1 \dots TA.i_n$ and irregular subscript expressions $e_1 \dots e_k$ with $TA.x_1 \dots TA.x_k$

11

Loop Restructuring and Iteration Reordering

- If a loop nest L containing irregular accesses is not perfectly nested, perform loop distribution to create perfectly nested loops.
- For each perfect loop nest containing irregular accesses, test independence of inner loop iterations.
- For each loop nest where the inner loops are parallel as well
 - Coalesce the loop nest into a single loop. The number of iterations are provided by the inspector.
 - Compute the source sets of the data accessed in each iteration by comparing the recorded subscript values with the reaching regular section descriptions
 - Sort iterations according to the access sets
 - Schedule the computation of an iteration set with the communication of data needed by another iteration set.

12

Creation of Computation and Communication Threads

- Non-blocking send/recv calls may not actually progress concurrently with computation.
 - Use a multi-threaded runtime system with separate computation and communication threads – on dual CPU machines these threads can progress concurrently.
- The compiler extracts the send/recvs along with the packing/unpacking of message buffers into a communication thread.
 - This enables a lean implementation of a multi-threaded MPI runtime library.

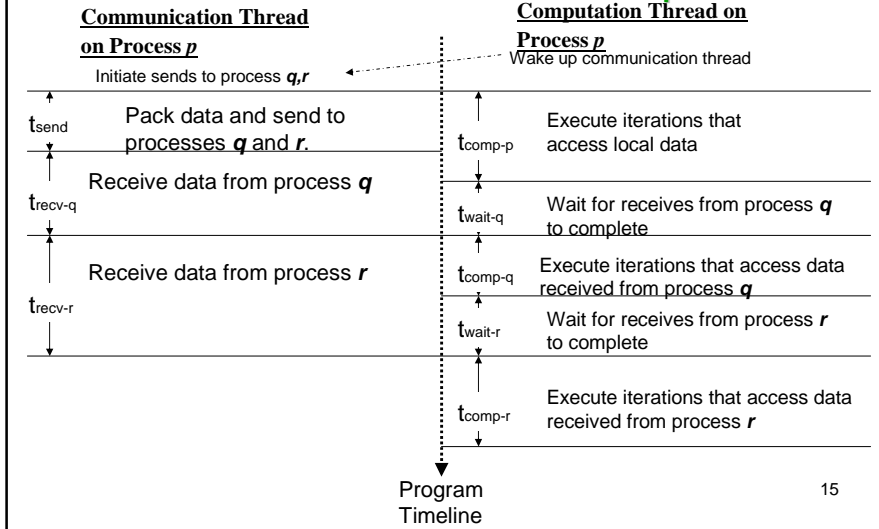
13

Experiments

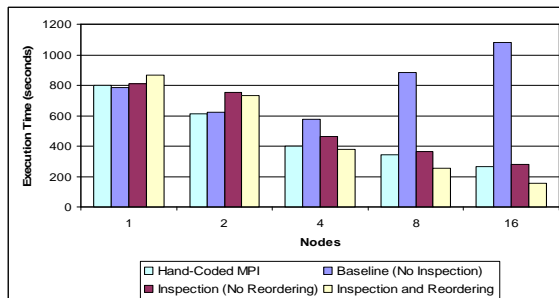
- Three representative irregular applications
 - **Equake** : Earthquake simulation from the SPEC OMPM2001 benchmark suite.
 - **Moldyn** : Molecular dynamics code from the CHARMM chemistry application.
 - **CG** : Conjugate Gradient kernel from the NAS Parallel Benchmark Suite
- Platform – Sixteen IBM SP2 WinterHawk-II dual-processor nodes, connected by a high-performance switch, with IBM MPI libraries and xlc version 6 (at optimization level O3).
- Evaluation – For each benchmark we compare the performance of the version translated using the techniques presented in this paper with
 - A hand-coded MPI version, that does not use computation-communication overlap.
 - A baseline version translated from OpenMP to MPI using only compile-time analysis.
 - A version translated from OpenMP to MPI that uses a run-time inspector, uses non-blocking communication between processes, but does not perform iteration reordering for computation-communication overlap.

14

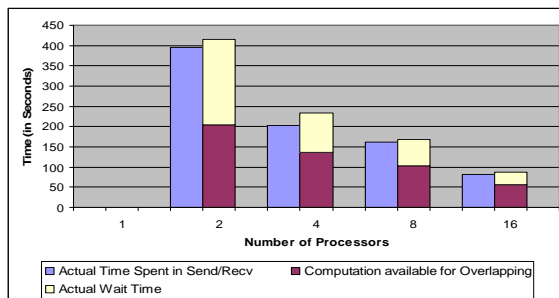
Measuring Computation-Communication Overlap



15

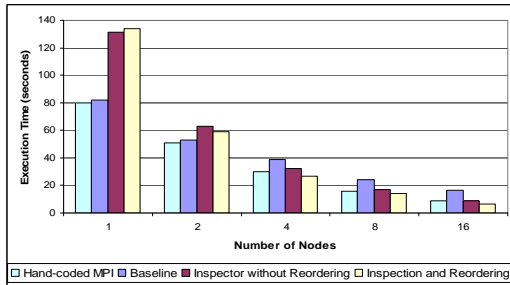


Performance of Equake

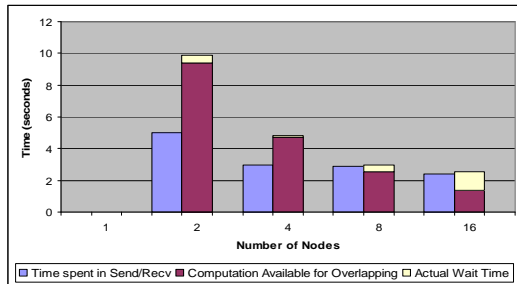


Computation-communication overlap in Equake

16

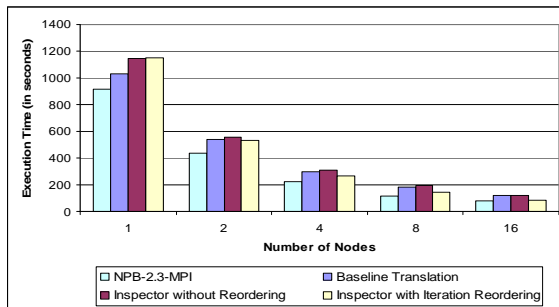


Performance of Moldyn

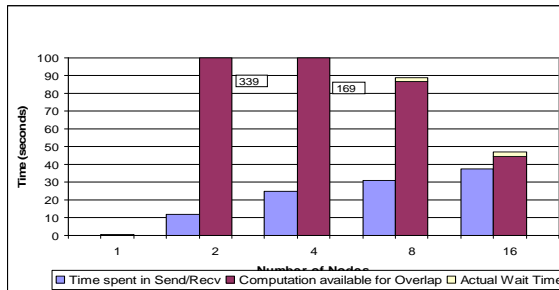


Computation-communication overlap in Moldyn

17



Performance of CG



Computation-communication overlap in CG

18

Related Work

- Compile-time and dynamic schemes for improving data locality
- Inspector/Executor models for HPF
- Computation-Communication Overlap schemes for HPF and Titanium
- Optimizations for Irregular Applications on Software Distributed Shared Memory Systems
 - We do additional loop restructuring and computation-communication overlap.

19

Summary

- We facilitate parallel programming productivity by enabling OpenMP programming for distributed-memory systems.
- We present techniques to optimize irregular OpenMP applications deployed on distributed-memory systems, by using inspection and loop restructuring for computation communication overlap.
- We evaluate our techniques for three applications and achieve speedups that are almost twice as fast as baseline OpenMP-to-MPI versions, almost 30% faster than inspector-only versions and within 9% of hand-coded MPI versions.

20

Questions ?

