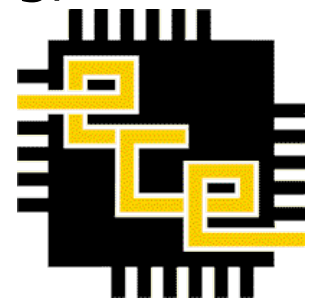


Programming Distributed Memory Systems Using OpenMP



Rudolf Eigenmann,
Ayon Basumallik, Seung-Jai Min,
School of Electrical and Computer Engineering,
Purdue University,
<http://www.ece.purdue.edu/ParaMount>





Is OpenMP a useful programming model for distributed systems?

- OpenMP is a parallel programming model that assumes a shared address space

```
#pragma OMP parallel for
for (i=1; 1<n; i++) {a[i] = b[i];}
```
- Why is it difficult to implement OpenMP for distributed processors?

The compiler or runtime system will need to

 - partition and place data onto the distributed memories
 - send/receive messages to orchestrate remote data accesses

HPF (High Performance Fortran) was a large-scale effort to do so - without success
- So, why should we try (again)?

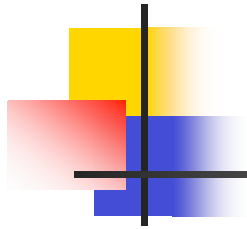
OpenMP is an easier programming (higher-productivity?) programming model. It

 - allows programs to be incrementally parallelized starting from the serial versions,
 - relieves the programmer of the task of managing the movement of logically shared data.



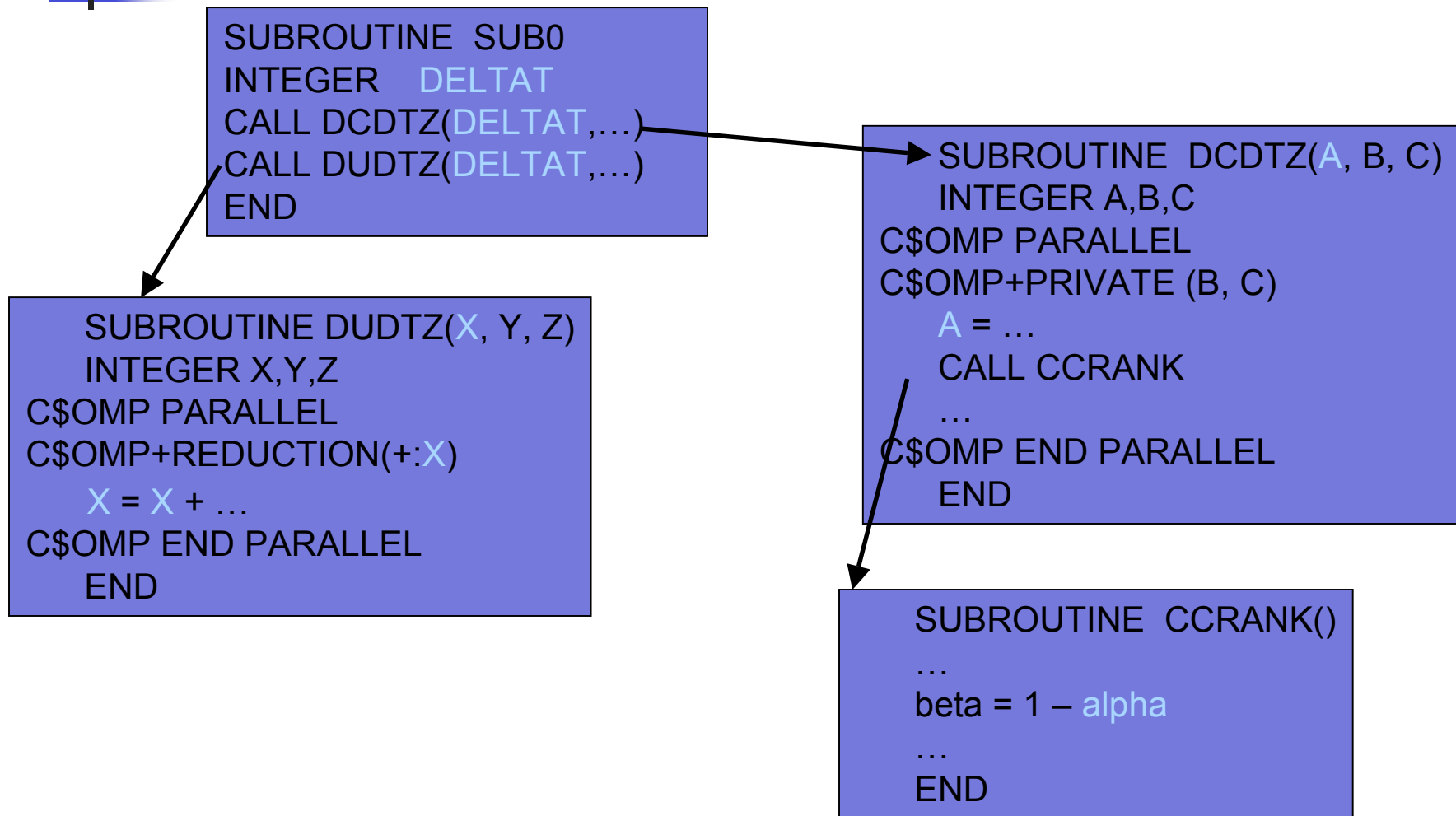
Two Translation Approaches

- Use a Software Distributed Shared Memory System
- Translate OpenMP directly to MPI



Approach 1: Compiling OpenMP for Software Distributed Shared Memory

Inter-procedural Shared Data Analysis



Access Pattern Analysis

```

DO istep = 1, itmax, 1

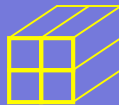
!$OMP PARALLEL DO
  rsd (i, j, k) = ...
!$OMP END PARALLEL DO

!$OMP PARALLEL DO
  rsd (i, j, k) = ...
!$OMP END PARALLEL DO

!$OMP PARALLEL DO
  u (i, j, k) = rsd (i, j, k)
!$OMP END PARALLEL DO

CALL RHS()

ENDDO
  
```



SUBROUTINE RHS()

```

!$OMP PARALLEL DO
  u (i, j, k) = ...
!$OMP END PARALLEL DO
  
```



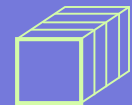
```

!$OMP PARALLEL DO
  ... = u (i, j, k)..
  rsd (i, j, k) = rsd (i, j, k)..
!$OMP END PARALLEL DO
  
```



```

!$OMP PARALLEL DO
  ... = u (i, j, k)..
  rsd (i, j, k) = rsd (i, j, k)..
!$OMP END PARALLEL DO
  
```



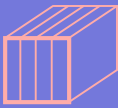
```

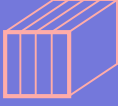
!$OMP PARALLEL DO
  ... = u (i, j, k)..
  rsd (i, j, k) = ...
!$OMP END PARALLEL DO
  
```




=> Data Distribution-Aware Optimization

```
DO istep = 1, itmax, 1
```

```
!$OMP PARALLEL DO   
  rsd (i, j, k) = ...  
!$OMP END PARALLEL DO
```

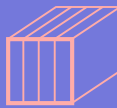
```
!$OMP PARALLEL DO   
  rsd (i, j, k) = ...  
!$OMP END PARALLEL DO
```


```
!$OMP PARALLEL DO   
  u (i, j, k) = rsd (i, j, k)  
!$OMP END PARALLEL DO
```


```
CALL RHS()
```


```
ENDDO
```

```
SUBROUTINE RHS()
```

```
!$OMP PARALLEL DO  
  u (i, j, k) = ...  
!$OMP END PARALLEL DO 
```

```
!$OMP PARALLEL DO  
  ... = u (i, j, k)..  
  rsd (i, j, k) = rsd (i, j, k)..  
!$OMP END PARALLEL DO 
```

```
!$OMP PARALLEL DO  
  ... = u (i, j, k)..  
  rsd (i, j, k) = rsd (i, j, k)..  
!$OMP END PARALLEL DO 
```

```
!$OMP PARALLEL DO  
  ... = u (i, j, k)..  
  rsd (i, j, k) = ...  
!$OMP END PARALLEL DO 
```

Adding Redundant Computation to Eliminate Communication

OpenMP Program

```
DO k = 1, z
!$OMP PARALLEL DO
  DO j = 1, N, 1
    flux(m, j) = u(3, i, j, k) + ...
  ENDDO
!$OMP PARALLEL DO
DO j = 1, N, 1
  DO m = 1, 5, 1
    rsd(m, i, j, k) = ... +
      flux(m, j+1)-flux(m, j-1))
  ENDDO
ENDDO
ENDDO
```

Optimized S-DSM Code S-DSM Program

```
init00 = (N/proc_num)*(pid-1)...
limit00 = (N/proc_num)*pid...
new_init = init00 - 1
new_limit = limit00 + 1
DO k = 1, z
  DO j = new_init, new_limit, 1
    flux(m, j) = u(3, i, j, k) + ...
  ENDDO
CALL TMK_BARRIER(0)
DO j = init00, limit00, 1
  DO m = 1, 5, 1
    rsd(m, i, j, k) = ... +
      flux(m, j+1)-flux(m, j-1))
  ENDDO
ENDDO
ENDDO
```


Access Privatization

Example from earthquake (SPEC OMPM2001)

```
If (master) {
  shared->ARCHnodes = .....
  shared->ARCHduration = ...
  ...
}

/* Parallel Region */
N = shared->ARCHnodes;
iter = shared->ARCHduration;
.....
```

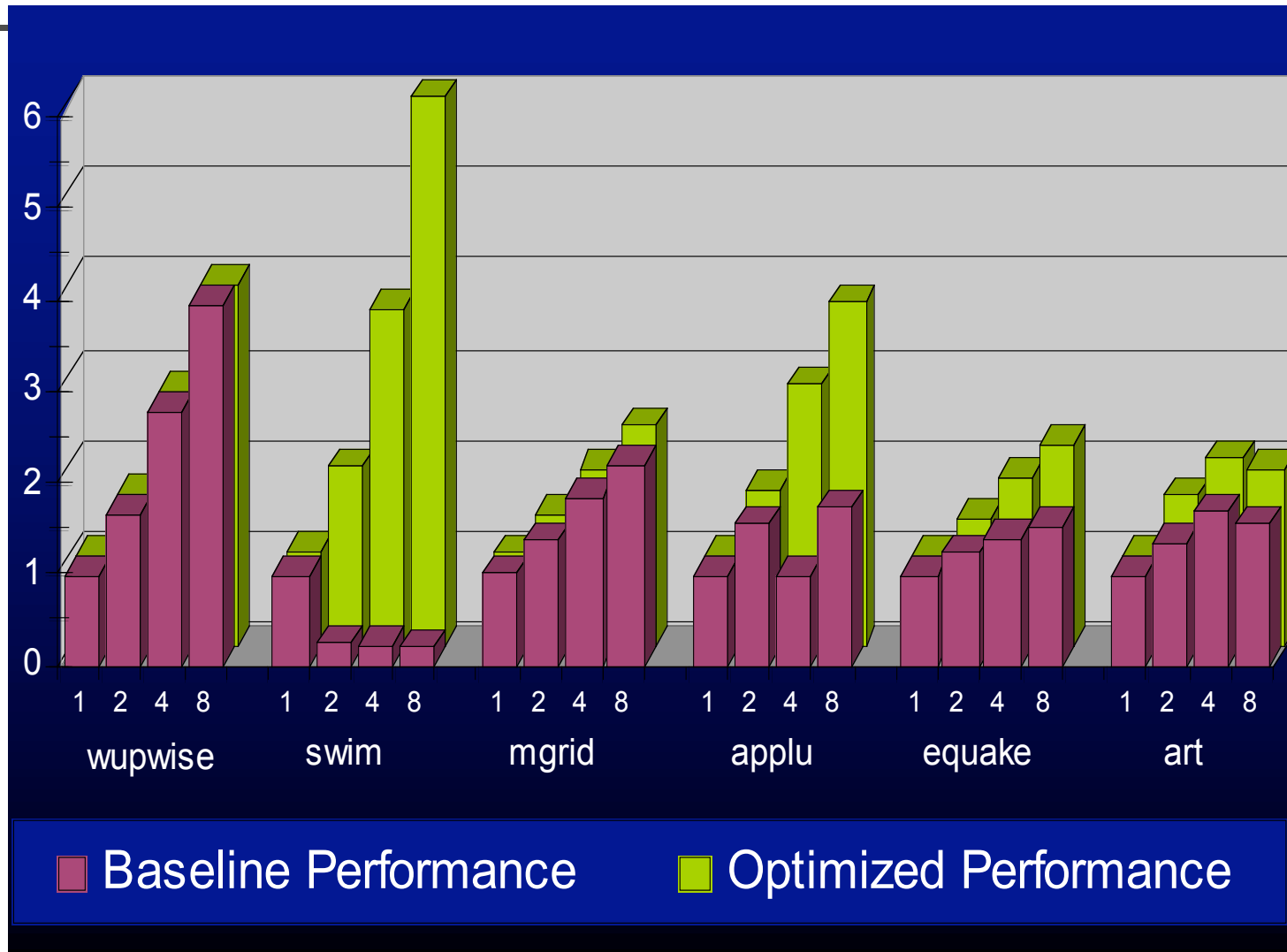
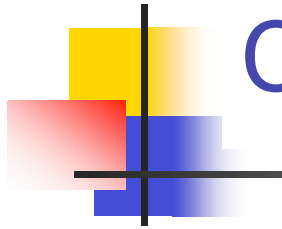
READ-ONLY SHARED VARS

```
// Done by all nodes
{ ARCHnodes = .....
  ARCHduration = ...
  ...
}

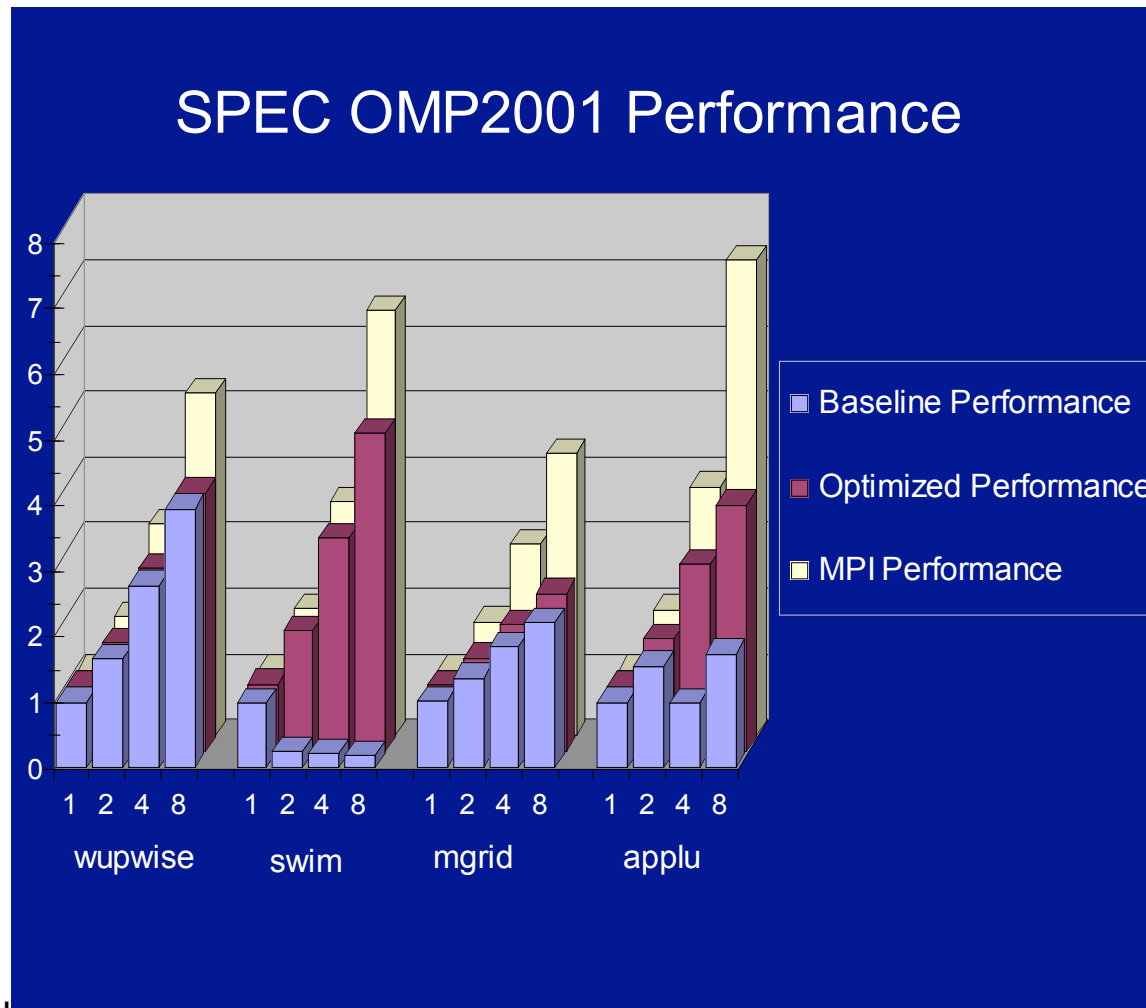
/* Parallel Region */
N = ARCHnodes;
iter = ARCHduration;
.....
```

PRIVATE VARIABLES

Optimized Performance of OMPM2001 Benchmarks



A Key Question: How Close Are we to MPI Performance ?

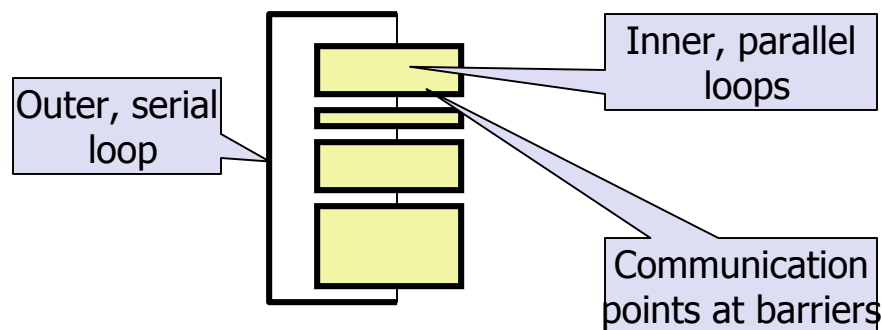


Towards Adaptive Optimization

A combined Compiler-Runtime Scheme

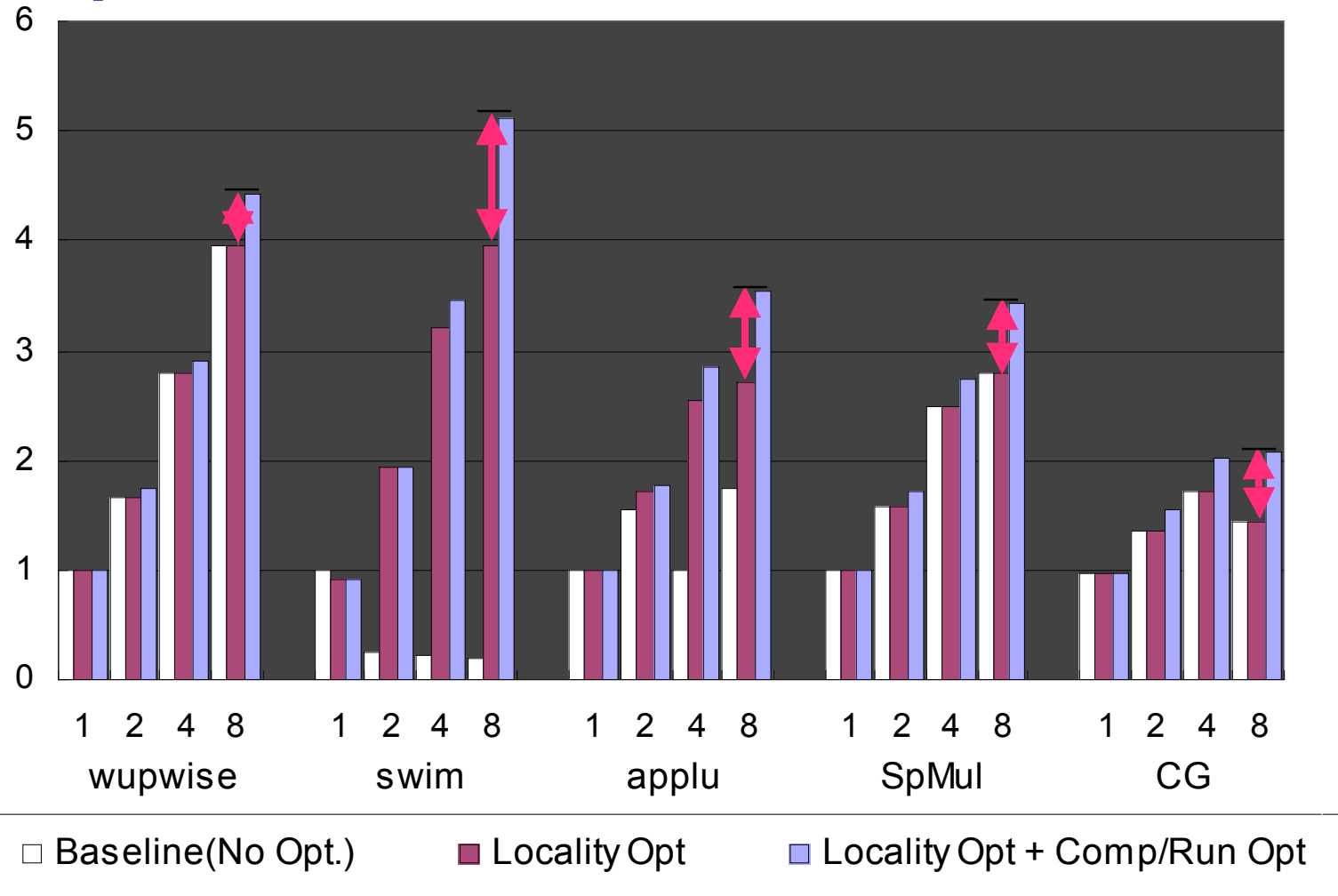
- Compiler identifies repetitive access patterns
- Runtime system learns the actual remote addresses and sends data early.

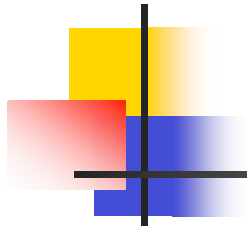
Ideal program characteristics:



Data addresses are invariant or a linear sequence, w.r.t. outer loop

Current Best Performance of OpenMP for S-DSM





Approach 2: Translating OpenMP directly to MPI

- Baseline translation
- Overlapping computation and communication for irregular accesses



Baseline Translation of OpenMP to MPI

- Execution Model
 - SPMD model
 - Serial Regions are replicated on all processes
 - Iterations of parallel **for** loops are distributed (using static block scheduling)
 - Shared Data is allocated on all nodes
 - There is no concept of “owner” – only producers and consumers of shared data
 - At the end of a parallel loop, producers communicate shared data to “potential” future consumers
 - Array section analysis is used for summarizing array accesses



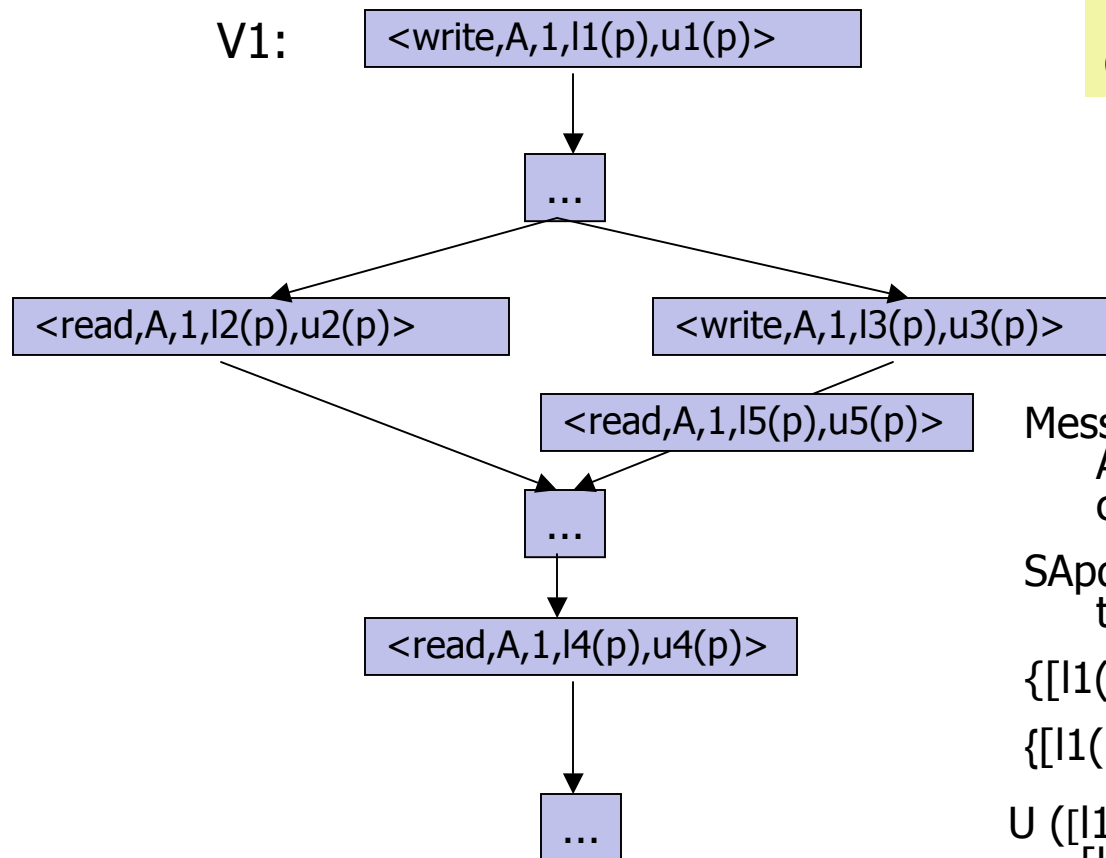
Baseline Translation

Translation Steps:

1. Identify all shared data
2. Create annotations for accesses to shared data (use regular section descriptors to summarize array accesses)
3. Use interprocedural data flow analysis to identify *potential consumers*; incorporate OpenMP relaxed consistency specifications
4. Create message sets to communicate data between producers and consumers

Message Set Generation

For every write,
determine all future reads



Message Set at RSD vertex V1, for array A from process p to process q computed as

S_{Apq} = Elements of A with subscripts in the set

$\{[l_1(p), u_1(p)] \cap [l_2(q), u_2(q)]\} \cup$

$\{[l_1(p), u_1(p)] \cap [l_4(q), u_4(q)]\}$

$\cup ([l_1(p), u_1(p)] \cap \{[l_5(q), u_5(q)] - [l_3(p), u_3(p)]\})$

Baseline Translation of Irregular Accesses



- Irregular Access – $A[B[i]]$, $A[f(i)]$
 - Reads: assumed the whole array accessed
 - Writes: inspect at runtime, communicate at the end of parallel loop
- We often can do better than “conservative”:
 - Monotonic array values \Rightarrow sharpen access regions

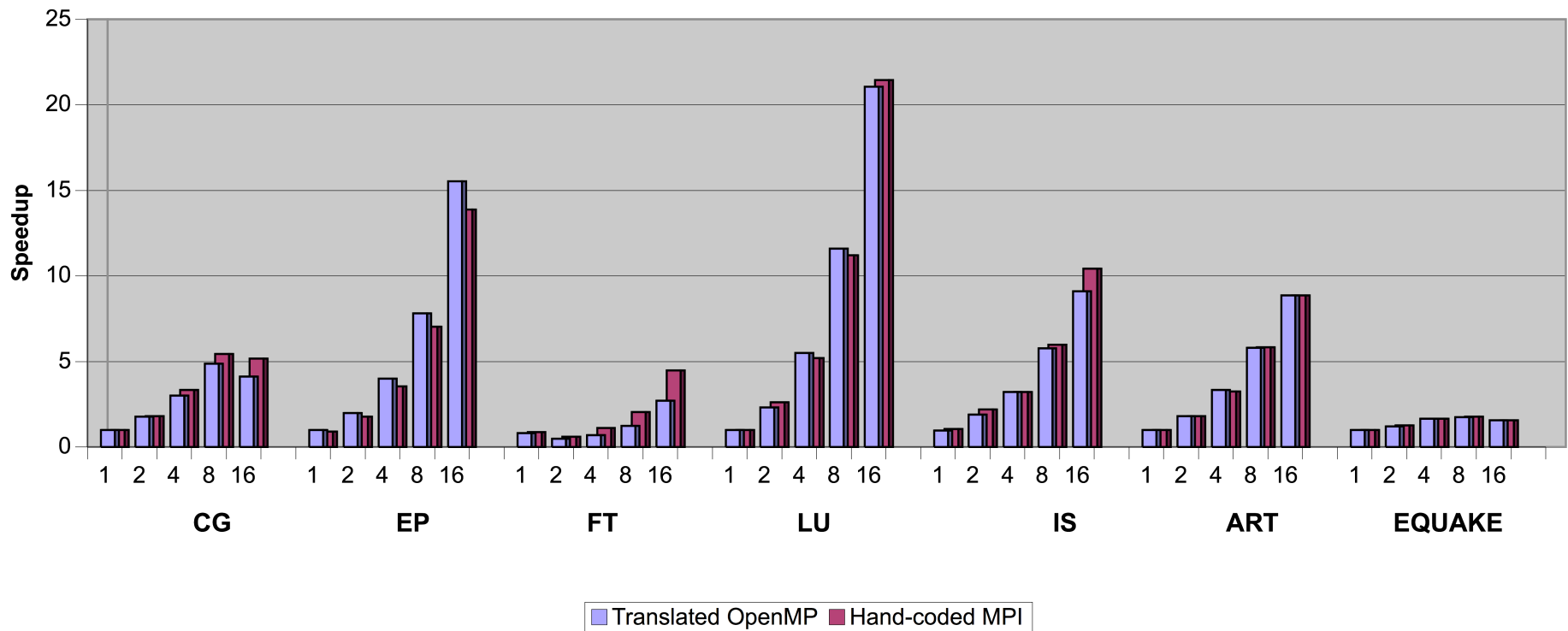


Optimizations based on Collective Communication

- Recognition of Reduction Idioms
 - Translate to MPI_Reduce / MPI_Allreduce functions.
- Casting sends/receives in terms of *alltoall* calls
 - Beneficial where the producer-consumer relationship is many-to-many and there is insufficient distance between producers and consumers.

Performance of the Baseline OpenMP to MPI Translation

Platform II – Sixteen IBM SP-2 WinterHawk-II nodes connected by a high-performance switch.

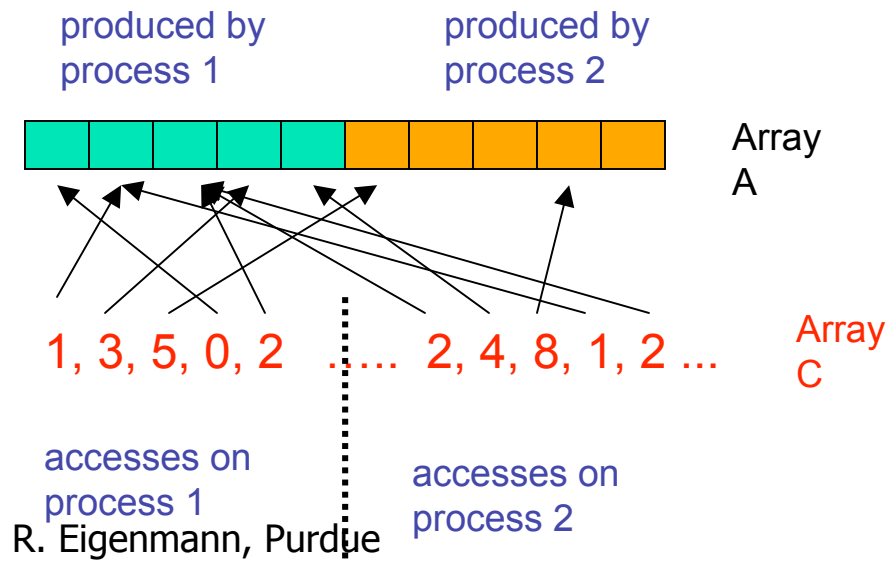


We can do more for Irregular Applications ?

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

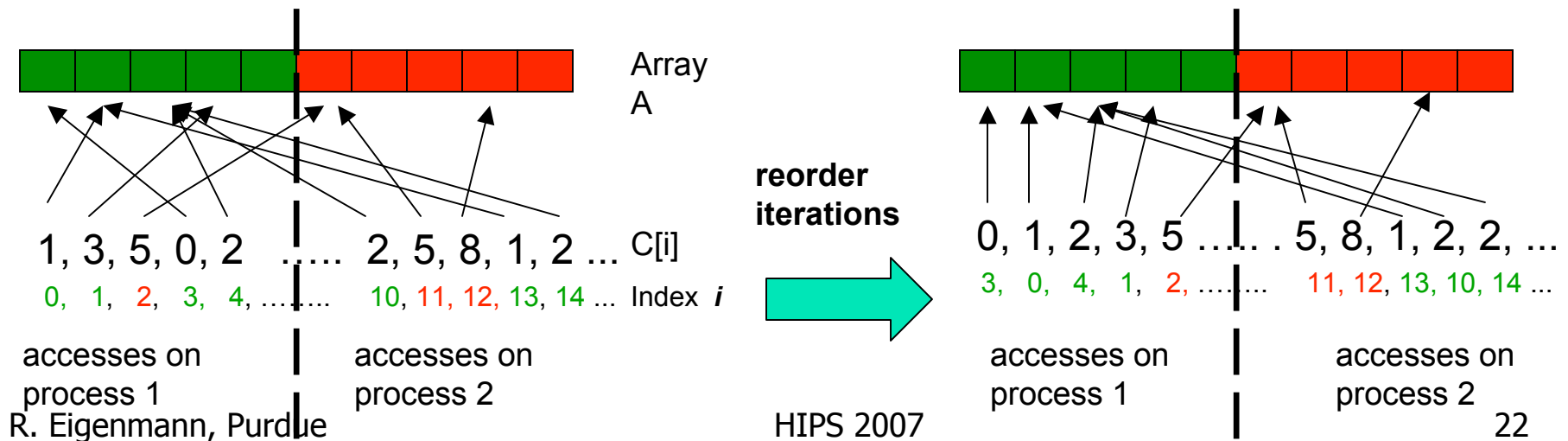
```
L2 : #pragma omp parallel for
      for(j=0;j<20;j++)
        B[j] = A[C[j]] + ...
```

- 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



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)



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
R. Eigenmann, Purdue

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

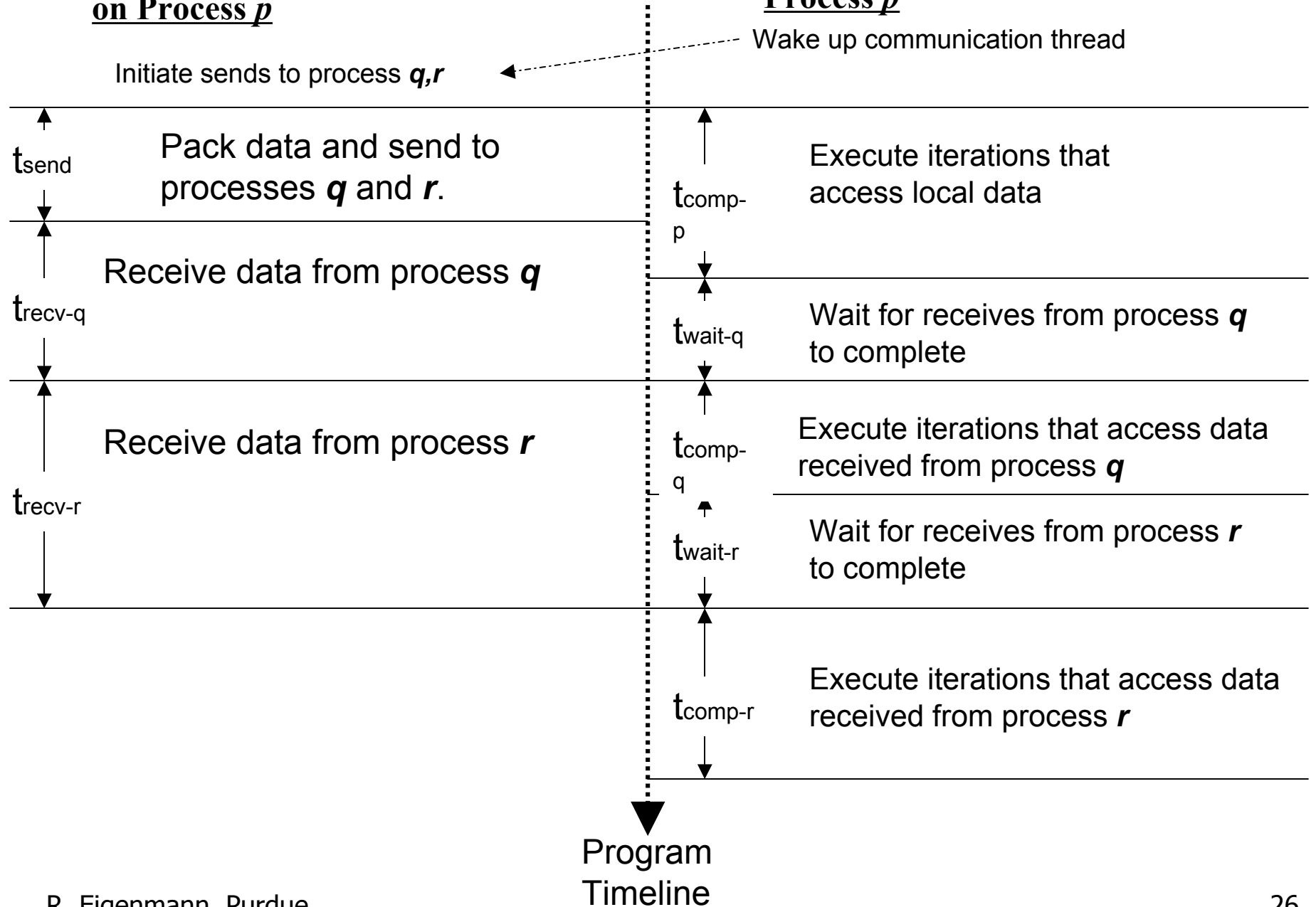


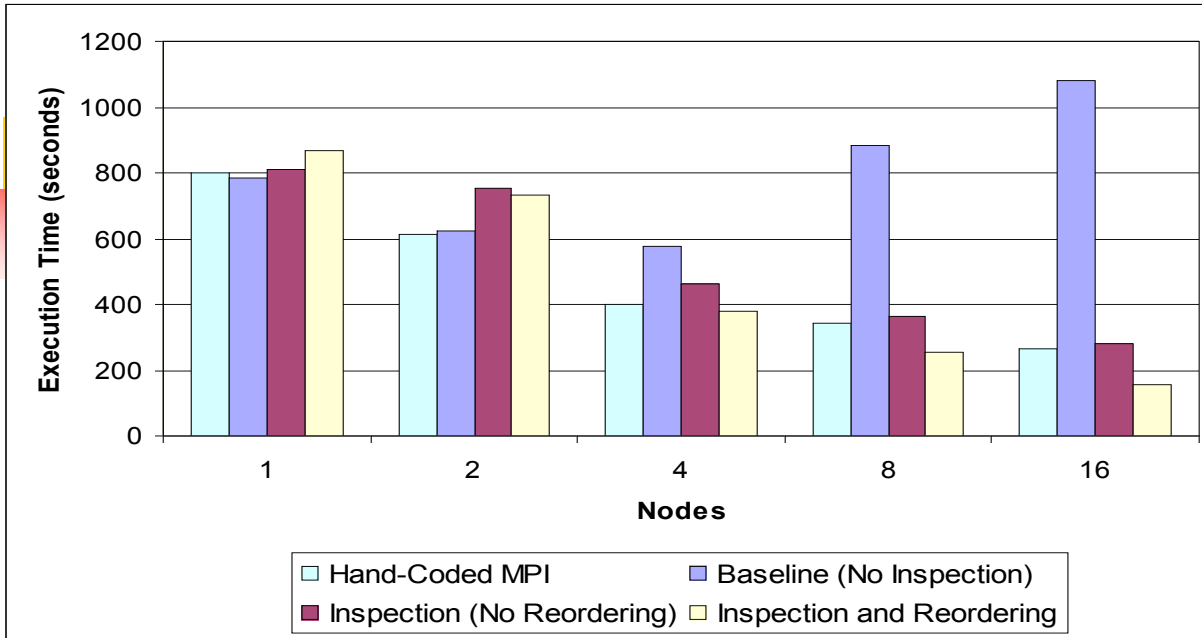
Achieving actual overlap of computation and communication

- 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.

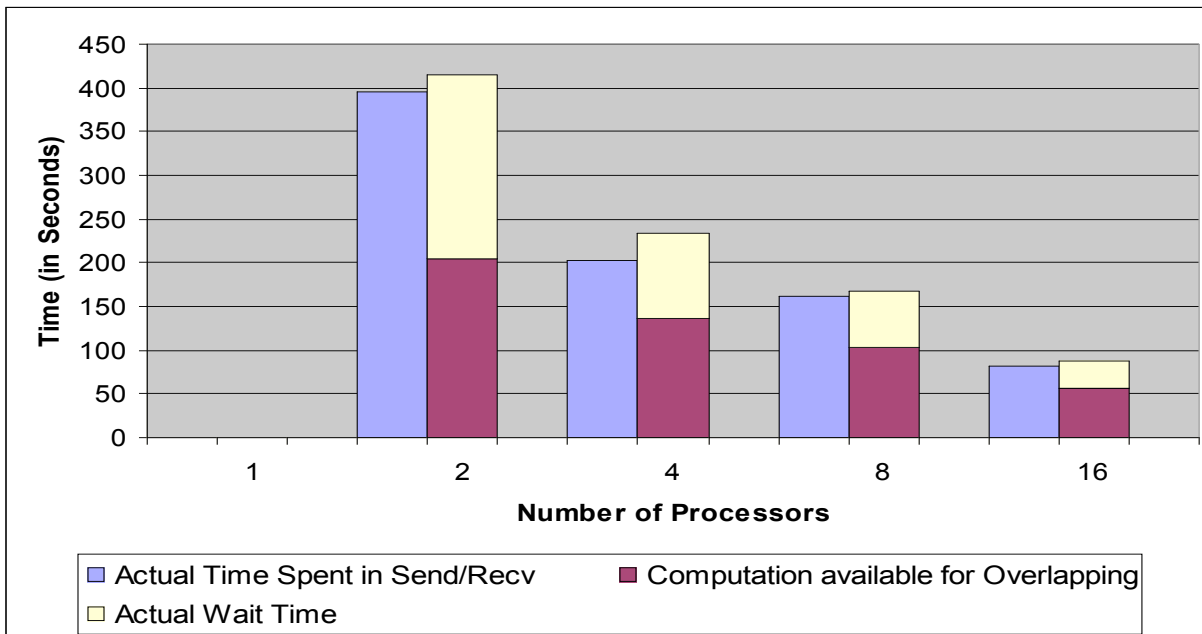
Communication Thread
on Process p

Computation Thread on
Process p

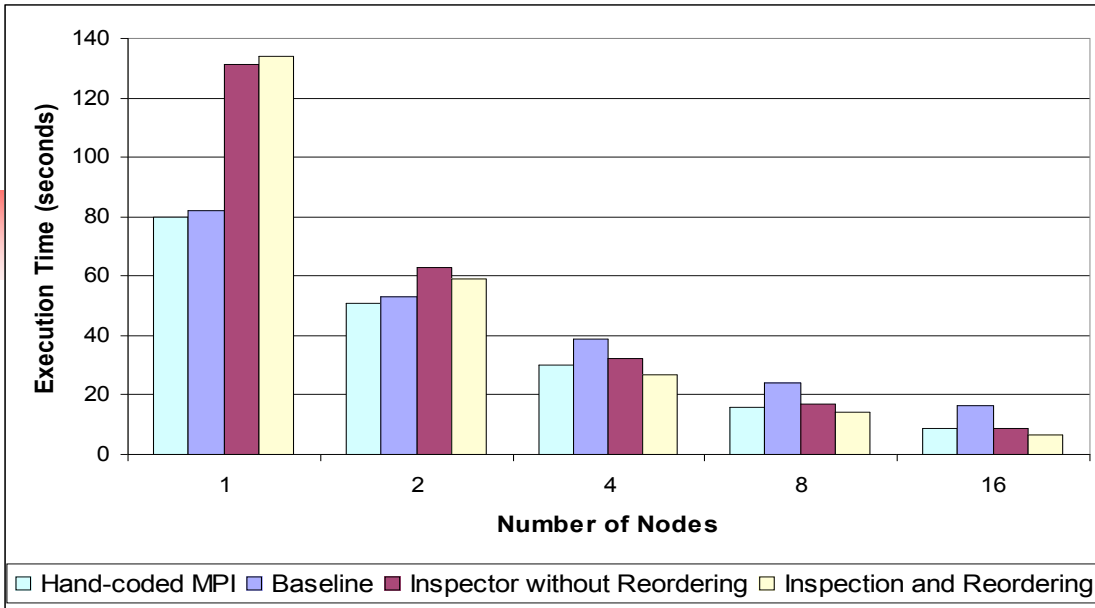




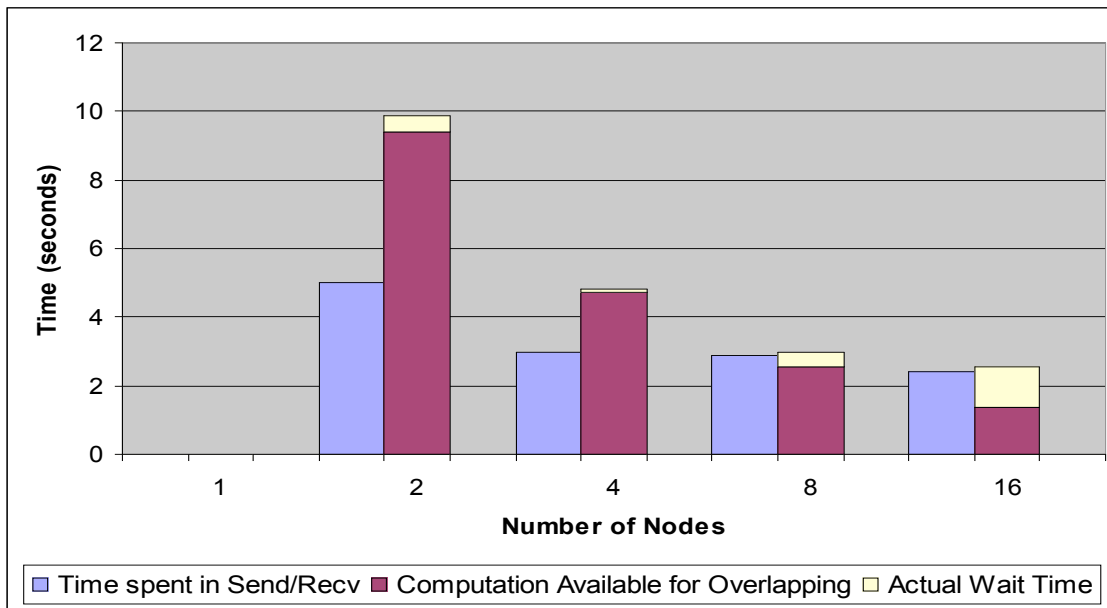
Performance of
Equake



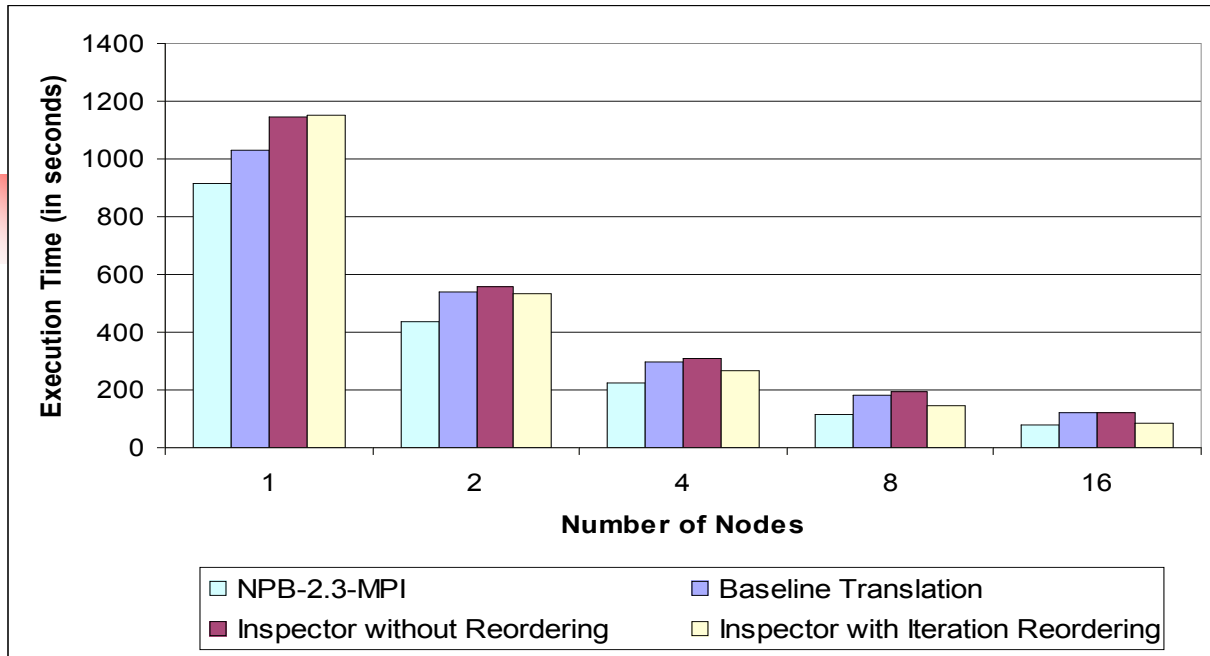
Computation-
communication
overlap in Equake



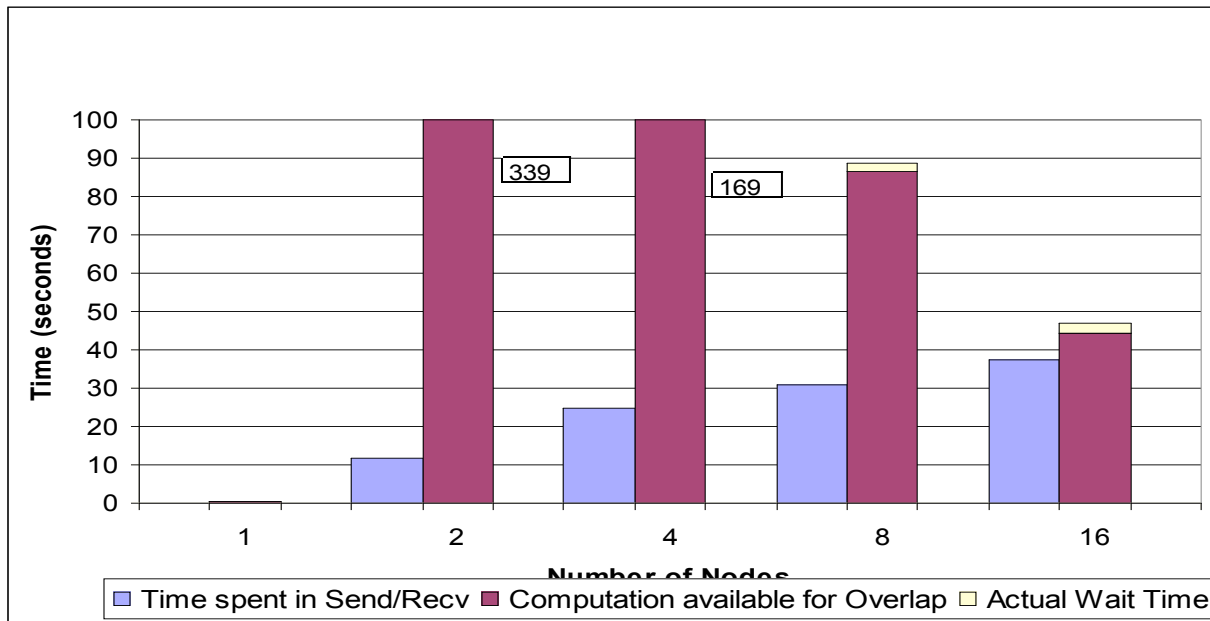
Performance of Moldyn



Computation-communication overlap in Moldyn



Performance of CG



Computation-communication overlap in CG



Conclusions

- There is hope for easier programming models on distributed systems
- OpenMP can be translated effectively onto DPS; we have used benchmarks from
 - SPEC OMP
 - NAS
 - additional irregular codes
- Direct Translation of OpenMP to MPI outperforms translation via S-DSM
 - “Fall back” of S-DSM for irregular accesses incurs significant overhead
- Caveats:
 - Data scalability is an issue
 - Black-belt programmers will always be able to do better
 - Advanced compiler technology is involved. There will be performance surprises.