Chapter 18

Combining MPI and OpenMP
Outline

• Advantages of using both MPI and OpenMP
• Case Study: Conjugate gradient method
• Case Study: Jacobi method
C+MPI vs. C+MPI+OpenMP

C + MPI

Interconnection Network

C + MPI + OpenMP

Interconnection Network
Why C + MPI + OpenMP Can Execute Faster

• Lower communication overhead
• More portions of program may be practical to parallelize
• May allow more overlap of communications with computations
Case Study: Conjugate Gradient

• Conjugate gradient method solves $Ax = b$

• In our program we assume $A$ is dense

• Methodology
  – Start with MPI program
  – Profile functions to determine where most execution time spent
  – Tackle most time-intensive function first
### Result of Profiling MPI Program

<table>
<thead>
<tr>
<th>Function</th>
<th>1 CPU</th>
<th>8 CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix_vector_product</td>
<td>99.55%</td>
<td>97.49%</td>
</tr>
<tr>
<td>dot_product</td>
<td>0.19%</td>
<td>1.06%</td>
</tr>
<tr>
<td>cg</td>
<td>0.25%</td>
<td>1.44%</td>
</tr>
</tbody>
</table>

Clearly our focus needs to be on function `matrix_vector_product`.
void matrix_vector_product (int id, int p, int n, double **a, double *b, double *c) {
    int i, j;
    double tmp; /* Accumulates sum */
    for (i=0; i<BLOCK_SIZE(id,p,n); i++) {
        tmp = 0.0;
        for (j = 0; j < n; j++)
            tmp += a[i][j] * b[j];
        piece[i] = tmp;
    }
    new_replicate_block_vector (id, p, piece, n, (void *) c, MPI_DOUBLE);
}
Adding OpenMP directives

• Want to minimize fork/join overhead by making parallel the outermost possible loop

• Outer loop may be executed in parallel if each thread has a private copy of tmp and j

```
#pragma omp parallel for private(j,tmp)
for (i=0; i<BLOCK_SIZE(id,p,n); i++) {
```

User Control of Threads

- Want to give user opportunity to specify number of active threads per process
- Add a call to `omp_set_num_threads` to function main
- Argument comes from command line

```c
omp_set_num_threads (atoi(argv[3]));
```
What Happened?

- We transformed a C+MPI program to a C+MPI+OpenMP program by adding only two lines to our program!
Benchmarking

• Target system: a commodity cluster with four dual-processor nodes
• C+MPI program executes on 1, 2, …, 8 CPUs
• On 1, 2, 3, 4 CPUs, each process on different node, maximizing memory bandwidth per CPU
• C+MPI+OpenMP program executes on 1, 2, 3, 4 processes
• Each process has two threads
• C+MPI+OpenMP program executes on 2, 4, 6, 8 threads
Results of Benchmarking

![Graph showing time (msec) vs. number of processors for MPI and MPI + OpenMP]
Analysis of Results

- C+MPI+OpenMP program slower on 2, 4 CPUs because C+MPI+OpenMP threads are sharing memory bandwidth, while C+MPI processes are not.
- C+MPI+OpenMP programs faster on 6, 8 CPUs because they have lower communication cost.
Case Study: Jacobi Method

• Begin with C+MPI program that uses Jacobi method to solve steady state heat distribution problem of Chapter 13

• Program based on rowwise block striped decomposition of two-dimensional matrix containing finite difference mesh
Methodology

- Profile execution of C+MPI program
- Focus on adding OpenMP directives to most compute-intensive function
## Result of Profiling

<table>
<thead>
<tr>
<th>Function</th>
<th>1 CPU</th>
<th>8 CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize_mesh</td>
<td>0.01%</td>
<td>0.03%</td>
</tr>
<tr>
<td>find_steady_state</td>
<td>98.48%</td>
<td>93.49%</td>
</tr>
<tr>
<td>print_solution</td>
<td>1.51%</td>
<td>6.48%</td>
</tr>
</tbody>
</table>
Function find_steady_state

(1/2)

its = 0;
for (; ;) {
    if (id > 0)
        MPI_Send (u[1], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD);
    if (id < p-1) {
        MPI_Send (u[my_rows-2], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD);
        MPI_Recv (u[my_rows-1], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD, &status);
    }
    if (id > 0)
        MPI_Recv (u[0], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD, &status);
Function find_steady_state (2/2)

diff = 0.0;
for (i = 1; i < my_rows-1; i++)
    for (j = 1; j < N-1; j++) {
        w[i][j] = (u[i-1][j] + u[i+1][j] +
                   u[i][j-1] + u[i][j+1])/4.0;
        if (fabs(w[i][j] - u[i][j]) > diff)
            diff = fabs(w[i][j] - u[i][j]);
    }
for (i = 1; i < my_rows-1; i++)
    for (j = 1; j < N-1; j++)
        u[i][j] = w[i][j];
MPI_Allreduce (&diff, &global_diff, 1,
               MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
if (global_diff <= EPSILON) break;
its++;
Making Function Parallel (1/2)

• Except for two initializations and a return statement, function is a big `for` loop

• Cannot execute `for` loop in parallel
  – Not in canonical form
  – Contains a `break` statement
  – Contains calls to MPI functions
  – Data dependences between iterations
Making Function Parallel (2/2)

• Focus on first for loop indexed by i
• How to handle multiple threads testing/updating diff?
• Putting if statement in a critical section would increase overhead and lower speedup
• Instead, create private variable tdiff
• Thread tests tdiff against diff before call to MPI_Allreduce
Modified Function

diff = 0.0;
#pragma omp parallel private (i, j, tdiff)
{
    tdiff = 0.0;
#pragma omp for
    for (i = 1; i < my_rows-1; i++)
        ...
#pragma omp for nowait
    for (i = 1; i < my_rows-1; i++)
#pragma omp critical
        if (tdiff > diff) diff = tdiff;
}

MPI_Allreduce (&diff, &global_diff, 1,
                MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
Making Function Parallel (3/3)

• Focus on second `for` loop indexed by `i`
• Copies elements of `w` to corresponding elements of `u`: no problem with executing in parallel
Benchmarking

- Target system: a commodity cluster with four dual-processor nodes
- C+MPI program executes on 1, 2, ..., 8 CPUs
- On 1, 2, 3, 4 CPUs, each process on different node, maximizing memory bandwidth per CPU
- C+MPI+OpenMP program executes on 1, 2, 3, 4 processes
- Each process has two threads
- C+MPI+OpenMP program executes on 2, 4, 6, 8 threads
Benchmarking Results

![Graph showing benchmarking results with time in seconds on the y-axis and number of processors on the x-axis. Two lines are plotted: one for MPI and one for MPI + OpenMP. As the number of processors increases, the time decreases for both methods.]
Analysis of Results

- Hybrid C+MPI+OpenMP program uniformly faster than C+MPI program
- Computation/communication ratio of hybrid program is superior
- Number of mesh points per element communicated is twice as high per node for the hybrid program
- Lower communication overhead leads to 19% better speedup on 8 CPUs
Summary

• Many contemporary parallel computers consist of a collection of multiprocessors.
• On these systems, performance of C+MPI+OpenMP programs can exceed performance of C+MPI programs.
• OpenMP enables us to take advantage of shared memory to reduce communication overhead.
• Often, conversion requires addition of relatively few pragmas.