OpenMP and MPI parallelization

Gundolf Haase

Institute for Mathematics and Scientific Computing
University of Graz, Austria

Geneve, Sept. 5, 2014
OpenMP for our example
OpenMP generation in code

- Determine matrix pattern and allocate memory for CRS
  
  \texttt{Get\_Matrix\_Pattern(nelem, 3, ia, nnz, id, ik, sk);}  
  
  remains sequential, only once needed

- Calculate Matrix entries and accumulate them
  
  \texttt{GetMatrix (nelem, 3, ia, nnode, xc, nnz, id, ik, sk, f);}  
  
  Parallel loop over all elements: \#pragma omp parallel for  
  \#pragma omp atomic needed in accumulation

- Apply Dirichlet boundary conditions
  
  \texttt{ApplyDirichletBC(nx, ny, neigh, u, id, ik, sk, f);}  
  
  remains sequential
Jacobi iteration

We solve $Ku = f$ by the Jacobi iteration ($\omega = 1$)

$$u^{k+1} := u^{k+1} + \omega D^{-1} \left( f - K \cdot u^k \right)$$

```cpp
JacobiSolve(nnode, id, ik, sk, f, u);
```

$$D := \text{diag}(K)$$

```cpp
D := diag(K) // #pragma omp parallel for
```

$$u := 0$$

```cpp
u := 0 // #pragma omp parallel for
```

$$r := f - K \cdot u^0$$

```cpp
r := f - K \cdot u^0 // #pragma omp parallel for
```

$$w := D^{-1} \cdot r$$

```cpp
w := D^{-1} \cdot r // #pragma omp parallel for
```

$$\sigma := \sigma_0 := (w, r)$$

```cpp
\sigma := \sigma_0 := (w, r) // #pragma omp parallel for reduction
```

$k := 0$

```cpp
k := 0
```

while $\sigma > \varepsilon^2 \cdot \sigma_0$ do

```cpp
while \sigma > \varepsilon^2 \cdot \sigma_0 do
```

$$k := k + 1$$

```cpp
k := k + 1 // #pragma omp parallel for
```

$$u^k := u^{k-1} + \omega \cdot w$$

```cpp
u^k := u^{k-1} + \omega \cdot w // #pragma omp parallel for
```

$$r := f - K \cdot u^k$$

```cpp
r := f - K \cdot u^k // #pragma omp parallel for
```

$$w := D^{-1} \cdot r$$

```cpp
w := D^{-1} \cdot r // #pragma omp parallel for
```

$$\sigma := (w, r)$$

```cpp
\sigma := (w, r) // #pragma omp parallel for reduction
```

end
Compile/Link: `g++ -fopenmp *.cpp -o main.GCC`

Set the number of parallel threads for the run:
`export OMP_NUM_THREADS 2`

run: `./main.GCC`

The number of threads can programmed into the code explicitly:
`omp_set_num_threads(2);` or via a clause in an OMP-pragma directive.

Code examples in `shm`. 
MPI for our example
### 6+6 basic functions in MPI

<table>
<thead>
<tr>
<th>Basic functions</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>MPI_Init</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>MPI_Finalize</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>MPI_Send</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>MPI_Recv</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>MPI_Comm_rank</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>MPI_Comm_size</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>MPI_Barrier</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>MPI_Bcast</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>MPI_Gather</td>
</tr>
<tr>
<td>MPI_Scatter</td>
<td>MPI_Scatter</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>MPI_Reduce</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>MPI_Allreduce</td>
</tr>
</tbody>
</table>
Start MPI

We only determine rank and number of processes.

```c
#include <mpi.h> // MPI

int main(int argc, char **argv)
{
    MPI_Comm icomm = MPI_COMM_WORLD; // take all MPI processes
    int myrank, numprocs; // my MPI-rank; number of MPI process
    MPI_Init(&argc,&argv); // start parallel MPI code
    MPI_Comm_rank(icomm, &myrank); // get my rank
    MPI_Comm_size(icomm, &numprocs); // get number of processes
    cout << "MPI process " << myrank << " of " << numprocs << endl;
    MPI_Barrier(icomm); flush(stdout); MPI_Barrier(icomm);
    MPI_Finalize(); // end parallel MPI code
    return 0;
}
```

mpicxx main.cpp -o main.GCC_
mpirun -np 2 ./main.GCC_
Point-to-point communication: Data exchange I

![Diagram of point-to-point communication with non-synchronized exchange]

**Figure**: Non-synchronized Exchange
A synchronized send `sSend` stops execution until the receiving process returns a receipt. If that process also waits for a receipt $\Rightarrow$ **dead lock**.
Collective operations: Gather and Scatter

Collect and distribute information from a root process to all processes (including the root itself)

Classically: The size of data for each process is the same
A pile of special gather/scatter operations exists also with individual data sizes
ALL versions exist where all processes function as a root.

Figure: Scatter and Gather
Broadcast

One root process send the identical data to all processes. This is just a special scatter.

**Figure**: 

![Broadcast operation diagram](image)
Reduce and Reduce-all operation

Individual data from the processes will be combined to a global result, available to root or to all processes.

Inner product: \[ \langle w, r \rangle = \sum_{i=1}^{P} \langle w_i, r_i \rangle \]

```cpp
#include <mpi.h> // MPI

float skalar(const int n, const float x[], const float y[], const MPI_Comm icomm) {
    const float s = dscapr(n,x,y); // call sequential inner product
    float sg;
    MPI_Allreduce(&s, &sg, 1, MPI_FLOAT, MPI_SUM, icomm);
    return (sg);
}
```

- You have to specify the data type (MPI_FLOAT) and the type of operation (MPI_SUM)
- An input array (s) and an output array (sg) have to be allocated.
- Here, the arrays have length 1.
MPI parallel Jacobi iteration

\[ D := \sum_{s=1}^{P} A_s^T \text{diag}(K_s) A_s \]  
// next neighbor comm.: \text{VecAccu}

\[ u := 0 \]

\[ r := f - K \cdot u^0 \]

\[ w := D^{-1} \cdot \sum_{s=1}^{P} A_s^T r_s \]  
// next neighbor comm.: \text{VecAccu}

\[ \sigma := \sigma_0 := (w, r) \]  
// parallel reduction: \text{MPI\_Allreduce}

\[ k := 0 \]

\[ \text{while } \sigma > \varepsilon^2 \cdot \sigma_0 \text{ do} \]

\[ k := k + 1 \]

\[ u^k := u^{k-1} + \omega \cdot w \]  
// no comm.

\[ r := f - K \cdot u^k \]  
// no comm.

\[ w := D^{-1} \cdot \sum_{s=1}^{P} A_s^T r_s \]  
// next neighbor comm.: \text{VecAccu}

\[ \sigma := (w, r) \]  
// parallel reduction: \text{MPI\_Allreduce}

end

See MPI-template code in \textit{par}, MPI solutions in \textit{Cxx.Solution}. 