

# OpenMP and MPI parallelization

Gundolf Haase

Institute for Mathematics and Scientific Computing  
University of Graz, Austria

Graz, Dec 2022



## OpenMP for our example

## OpenMP generation in code

- ▶ Determine matrix pattern and allocate memory for CRS

```
Get_Matrix_Pattern(nelem, 3, ia, nnz, id, ik, sk);
```

remains sequential, only once needed

- ▶ Calculate Matrix entries and accumulate them

```
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

```
ApplyDirichletBC(nx, ny, neigh, u, id, ik, sk, f);
```

remains sequential

## Jacobi iteration

We solve  $K\underline{u} = \underline{f}$  by the Jacobi iteration ( $\omega = 1$ )

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

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

```
D := diag(K)                                // #pragma omp parallel for
u := 0
r := f - K · u0
w := D-1 · r
σ := σ0 := (w, r)
k := 0
while σ > ε2 · σ0 do
    k := k + 1
    uk := uk-1 + ω · w                    // #pragma omp parallel for
    r := f - K · uk                        // #pragma omp parallel for
    w := D-1 · r                            // #pragma omp parallel for
    σ := (w, r)                            // #pragma omp parallel for reduction
end
```

# OpenMP compiling

- ▶ 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 be 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

### Basic functions

MPI\_Init  
MPI\_Finalize  
MPI\_Send  
MPI\_Recv  
MPI\_Comm\_rank  
MPI\_Comm\_size  
  
MPI\_Barrier  
MPI\_Bcast  
MPI\_Gather  
MPI\_Scatter  
MPI\_Reduce  
MPI\_Allreduce

# Start MPI

We only determine rank and number of processes.

```
1 #include <mpi.h>           // MPI
2
3 int main(int argc, char **argv)
4 {
5     MPI_Comm icomm = MPI_COMM_WORLD;    // take all MPI processes
6     int myrank, numprocs;               // my MPI-rank; number of MPI process
7
8     MPI_Init(&argc,&argv);               // start parallel MPI code
9
10    MPI_Comm_rank(icom, &myrank);        // get my rank
11    MPI_Comm_size(icom, &numprocs);      // get number of processes
12
13    cout << "MPI_process_" << myrank << "out_of_" << numprocs << endl;
14    MPI_Barrier(icom); fflush(stdout); MPI_Barrier(icom);
15
16    MPI_Finalize();                     // end parallel MPI code
17
18    return 0;
19 }
20
```

```
mpicxx main.cpp -o main.GCC_
```

```
mpirun -np 2 ./main.GCC_
```



## Point-to-point communication: Data exchange I

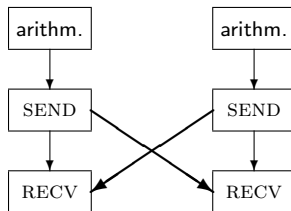


Figure: Non-synchronized EXCHANGE

## Point-to-point communication: Data exchange II

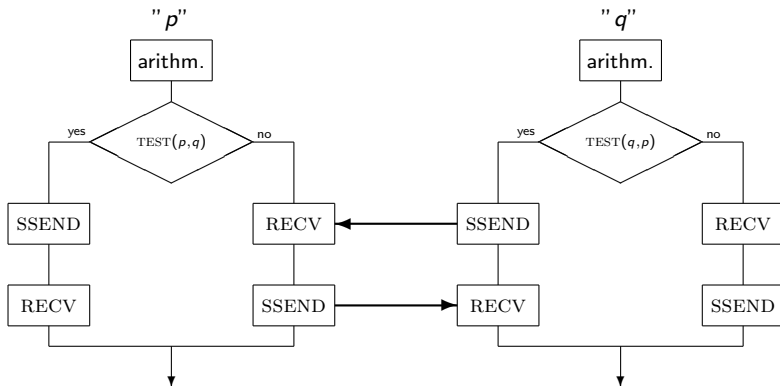


Figure: Synchronized EXCHANGE

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 process (including the root itself)

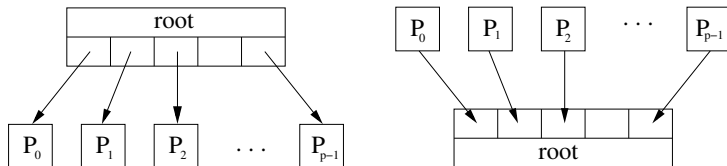
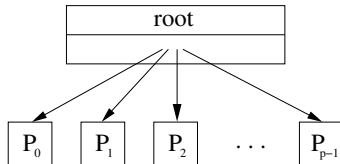


Figure: SCATTER and GATHER

- ▶ 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 a root.

## Broadcast

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



**Figure:** BROADCAST operation

## Reduce and Reduce-all operation

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

$$\text{Inner product: } \langle \mathbf{w}, \mathbf{r} \rangle = \sum_{i=1}^P \langle \mathbf{w}_i, \mathbf{r}_i \rangle$$

```
#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

```
 $\mathcal{D} := \sum_{s=1}^P A_s^T \text{diag}(K_s) A_s$  // next neighbor comm.: VecAccu  
 $\underline{u} := 0$   
 $\underline{r} := \underline{f} - K \cdot \underline{u}^0$   
 $\underline{w} := \mathcal{D}^{-1} \cdot \sum_{s=1}^P A_s^T \underline{r}_s$  // next neighbor comm.: VecAccu  
 $\sigma := \sigma_0 := (\underline{w}, \underline{r})$  // parallel reduction: MPI_Allreduce  
 $k := 0$   
while  $\sigma > \varepsilon^2 \cdot \sigma_0$  do  
     $k := k + 1$   
     $\underline{u}^k := \underline{u}^{k-1} + \omega \cdot \underline{w}$  // no comm.  
     $\underline{r} := \underline{f} - K \cdot \underline{u}^k$  // no comm.  
     $\underline{w} := \mathcal{D}^{-1} \cdot \sum_{s=1}^P A_s^T \underline{r}_s$  // next neighbor comm.: VecAccu  
     $\sigma := (\underline{w}, \underline{r})$  // parallel reduction: MPI_Allreduce  
end
```

See MPI-template code in *par*, MPI solutions in *Cxx.Solution*.