OpenMP and MPI parallelization

Gundolf Haase

Institute for Mathematics and Scientific Computing
University of Graz, Austria

Chile, Jan. 2015

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

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

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

$$
\begin{array}{lll}
D & := & \mathrm{diag}(K) \\
\underline{u} & := & 0 \\
\underline{r} & := & \underline{f} - K \cdot \underline{u}^0 \\
\underline{w} & := & D^{-1} \cdot \underline{r} \\
\sigma & := & \sigma_0 \;:=\; (\underline{w}, \underline{r}) \\
k & := & 0 \\
\text{while} & \sigma > \varepsilon^2 \cdot \sigma_0 & \text{do} \\
 & k & := & k+1 \\
 & \underline{u}^k & := & \underline{u}^{k-1} + \omega \cdot \underline{w} \\
 & \underline{r} & := & \underline{f} - K \cdot \underline{u}^k \\
 & \underline{w} & := & D^{-1} \cdot \underline{r} \\
 & \sigma & := & (\underline{w}, \underline{r}) \\
\text{end} & & & &
\end{array}
$$

```
// #pragma omp parallel for
// #pragma omp parallel for
// #pragma omp parallel for
// #pragma omp parallel for
// #pragma omp parallel for reduction
```

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 programmed into the code the code explicitely:
  `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.

```
#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 << "out of " << numprocs << endl;
  MPI_Barrier(icomm); fflush(stdout); MP_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

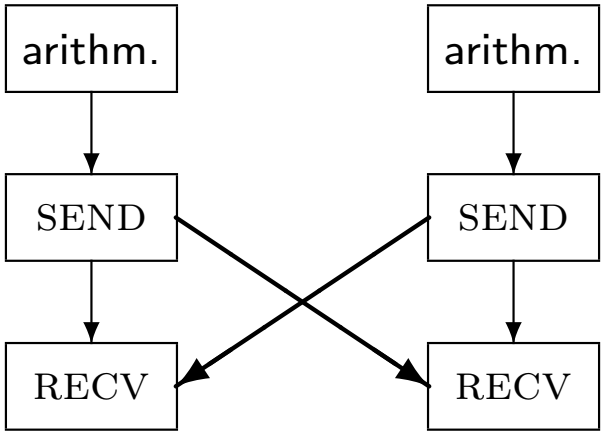

Figure : Non-synchronized EXCHANGE

Point-to-point communication: Data exchange II

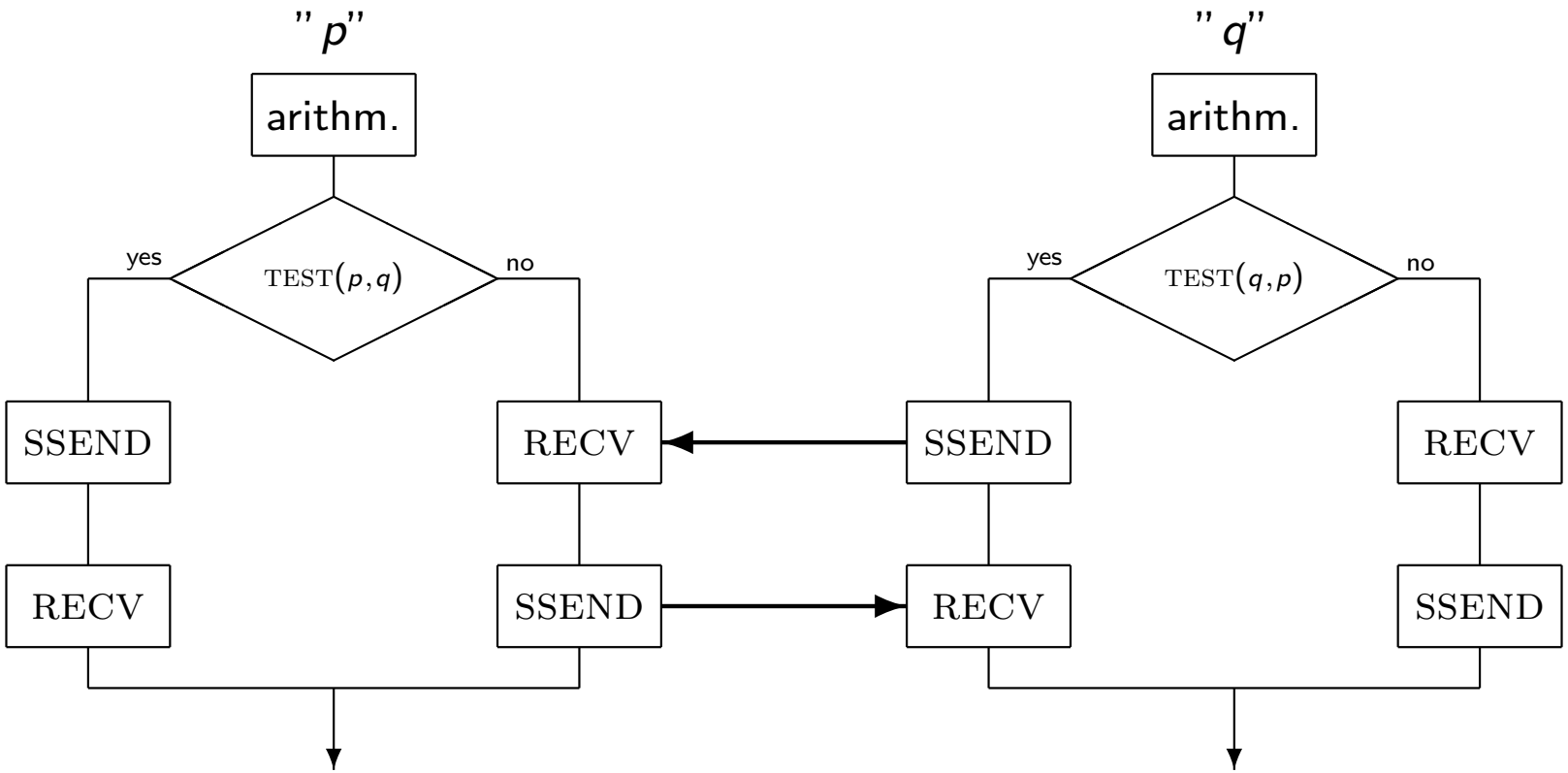

Figure : Synchronized EXCHANGE

A synchronized send SSEND stops execution until the receiving process returns a receipt. If that process also waits for a receipt $\Longrightarrow$ **dead lock**.

Collective operations: Gather and Scatter

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

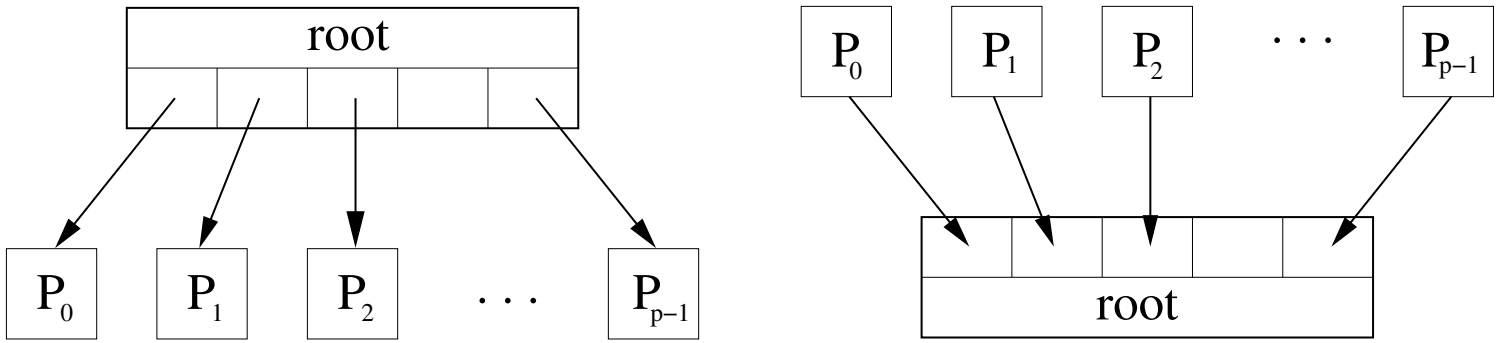

Figure : SCATTER and GATHER

- Classically: The size of data for eacxh 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 sroot.

Broadcast

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

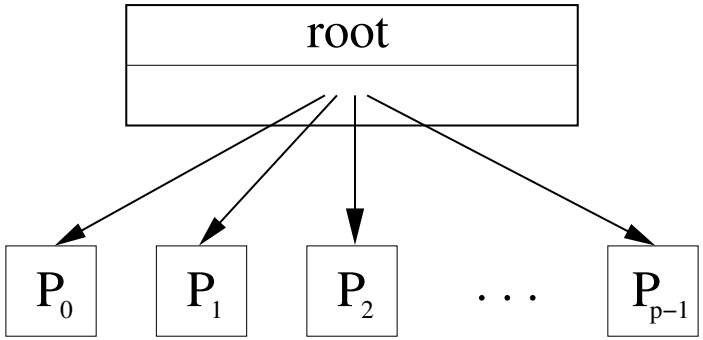

Figure : BROADCAST operation

Reduce and Reduce-all operation

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

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

```
#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

$$
\begin{aligned}
\mathfrak{D} &:= \sum_{s=1}^{P} A_s^T \operatorname{diag}(\mathsf{K}_s) A_s && \text{// next neighbor comm.: VecAccu} \\
\underline{\mathfrak{u}} &:= 0 \\
\underline{\mathsf{r}} &:= \underline{\mathsf{f}} - \mathsf{K} \cdot \underline{\mathfrak{u}}^0 \\
\underline{\mathfrak{w}} &:= \mathfrak{D}^{-1} \cdot \sum_{s=1}^{P} A_s^T \underline{\mathsf{r}}_s && \text{// next neighbor comm.: VecAccu} \\
\sigma &:= \sigma_0 := (\underline{\mathfrak{w}}, \underline{\mathsf{r}}) && \text{// parallel reduction: MPI\_Allreduce} \\
k &:= 0 \\
\text{while} \quad & \sigma > \varepsilon^2 \cdot \sigma_0 \quad \text{do} \\
& \quad k := k + 1 \\
& \quad \underline{\mathfrak{u}}^k := \underline{\mathfrak{u}}^{k-1} + \omega \cdot \underline{\mathfrak{w}} && \text{// no comm.} \\
& \quad \underline{\mathsf{r}} := \underline{\mathsf{f}} - \mathsf{K} \cdot \underline{\mathfrak{u}}^k && \text{// no comm.} \\
& \quad \underline{\mathfrak{w}} := \mathfrak{D}^{-1} \cdot \sum_{s=1}^{P} A_s^T \underline{\mathsf{r}}_s && \text{// next neighbor comm.: VecAccu} \\
& \quad \sigma := (\underline{\mathfrak{w}}, \underline{\mathsf{r}}) && \text{// parallel reduction: MPI\_Allreduce} \\
\text{end} &
\end{aligned}
$$

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