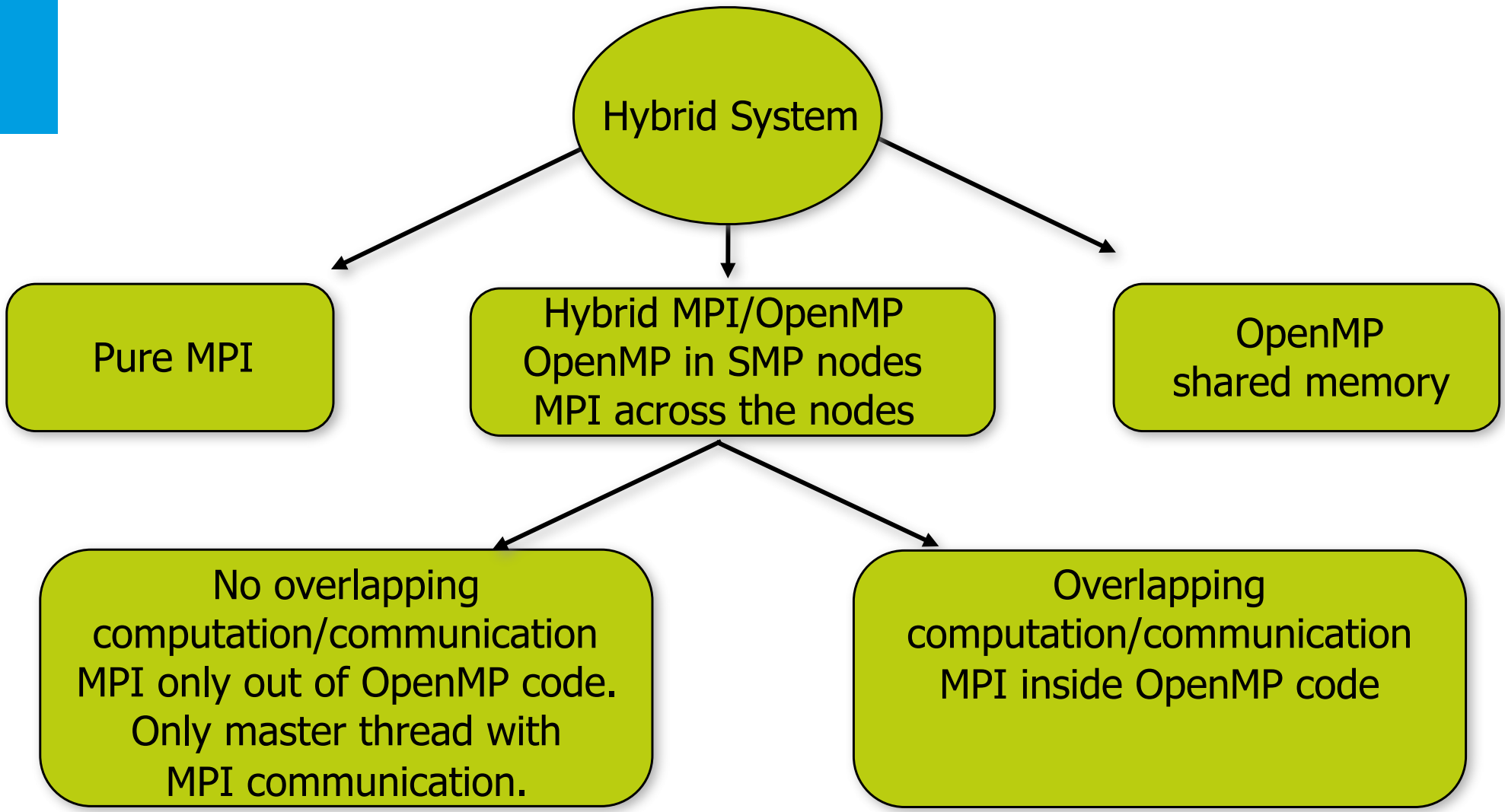
Programming with MPI

Hybrid MPI + OpenMP

Jan Thorbecke

Hybrid systems programming hierarchy



What Is OpenMP?

- Compiler directives for multithreaded programming
- Easy to create threaded Fortran and C/C++ codes
- Supports data parallelism model
- Portable and Standard
- Incremental parallelism
  - ➡Combines serial and parallel code in single source

Directive based

- Directives are special comments in the language
  - Fortran fixed form: `!$OMP, C$OMP, *$OMP`
  - Fortran free form: `!$OMP`

Special comments are interpreted by OpenMP compilers

```
      w = 1.0/n
      sum = 0.0
!$OMP PARALLEL DO PRIVATE(x) REDUCTION(+:sum)
      do I=1,n
        x = w*(I-0.5)
        sum = sum + f(x)
      end do
      pi = w*sum
      print *,pi
      end
```

Comment in Fortran
but interpreted by OpenMP compilers

C example

`#pragma omp` directives in C

– Ignored by non-OpenMP compilers

```
  w = 1.0/n;
  sum = 0.0;
#pragma omp parallel for private(x) reduction(+:sum)
  for(i=0, i<n, i++) {
    x = w*((double)i+0.5);
    sum += f(x);
  }
  pi = w*sum;
  printf("pi=%g\n", pi);
}
```

Data Environment

- OpenMP uses a shared-memory programming model
  - Most variables are shared by default.
  - Global variables are shared among threads
    C/C++: File scope variables, static
- Not everything is shared, there is often a need for "local" data as well

About Variables in SMP

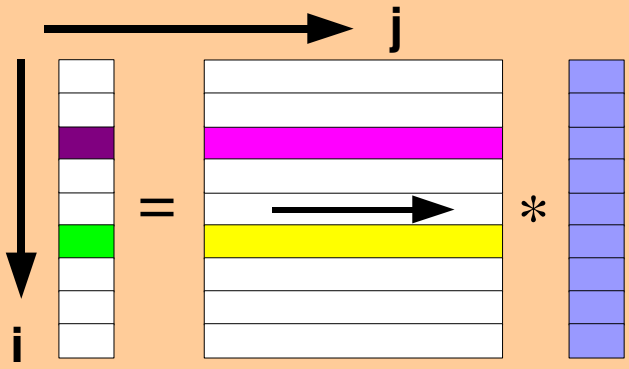
- Shared variables
  Can be accessed by every thread thread. Independent read/write operations can take place.

- Private variables
  Every thread has it's own copy of the variables that are created/ destroyed upon entering/leaving the procedure. They are not visible to other threads.

serial code	**parallel code**
global	shared
auto local	local
static	use with care
dynamic	use with care

Matrix-vector example

```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```



TID = 0

```
for (i=0,1,2,3,4)
 i = 0
sum = Σ  b[i=0][j]*c[j]
   a[0] = sum
 i = 1
sum = Σ  b[i=1][j]*c[j]
   a[1] = sum
```

TID = 1

```
for (i=5,6,7,8,9)
 i = 5
sum = Σ  b[i=5][j]*c[j]
   a[5] = sum
 i = 6
sum = Σ  b[i=6][j]*c[j]
   a[6] = sum
```

etc

Numerical Integration to Compute Pi

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;
   for (i=0; i<num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

Parallelize the numerical integration code using OpenMP

What variables can be shared?

step, num_steps

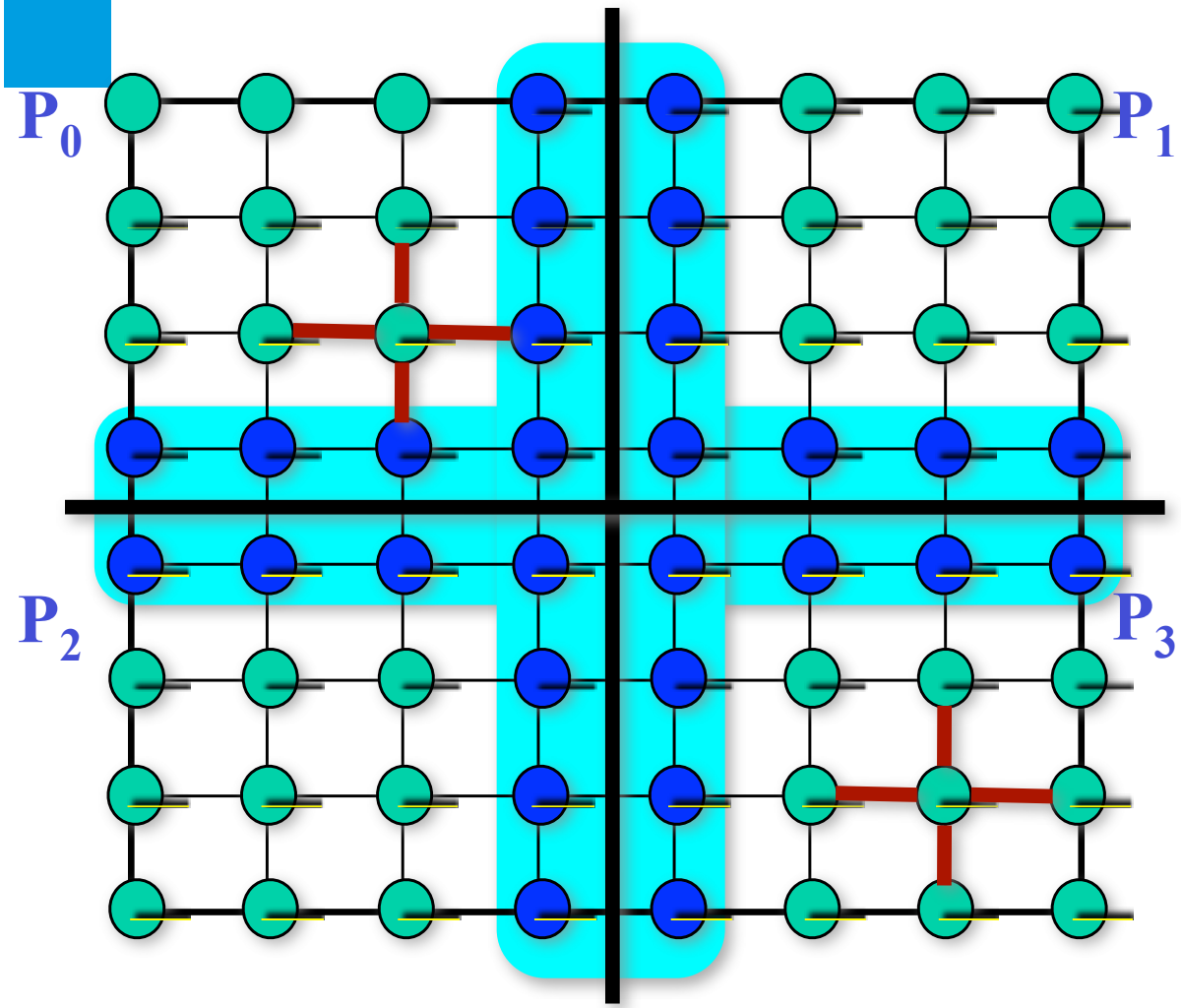
What variables need to be private?

x, i

What variables should be set up for reductions?

sum

Overlapping computation/communication: Example



Suppose we wish to solve the PDE

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x,y)$$

Using the Jacobi method: the value of u at each discretization point is given by a certain average among its neighbors, until convergence.

Distributing the mesh to SMP clusters by Domain Decomposition, it is clear that the GREEN nodes can proceed without any comm., while the Blue nodes have to communicate first and calculate later.

MPI/OpenMPI: Overlapping computation/ communication

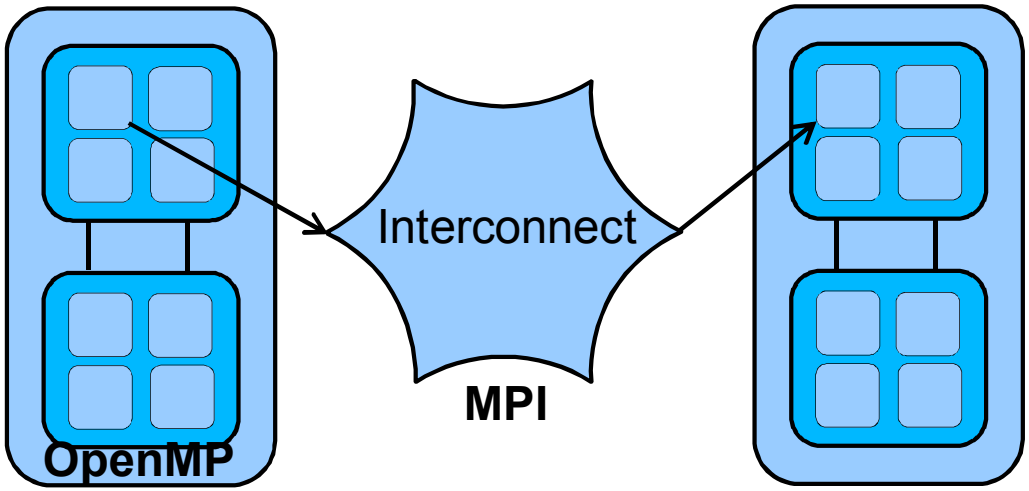
Not only the master but other threads communicate. Call MPI primitives in OpenMP code regions.

```
if (my_thread_id < # ){
    MPI_… (communicate needed data)
} else
    /* Perform computations that to not need
    communication */
    .
    .
}
/* All threads execute code that requires
    communication */
    .
    .
```

```
for (k=0; k < MAXITER; k++){
    /* Start parallel region here */
    #pragma omp parallel private(){
            my_id = omp_get_thread_num();

            if (my_id is given “halo points”)
                    MPI_SendRecv(“From neighboring MPI process”);
            else{
                    for (i=0; i < # allocated points; i++)
                            newval[i] = avg(oldval[i]);
            }

            if (there are still points I need to do) /* Thi
                    for (i=0; i< # remaining points; i++)
                            newval[i] = avg(oldval[i]);

            }
            for (i=0; i<(all_my_points); i++)
                            oldval[i] = newval[i];
    }
    MPI_Barrier(); /* Synchronize all MPI processes here */
}
```

Hybrid programming

- Parallel programming model combining:
  - Parallelization over one SMP node with shared-memory parallelization
  - Parallelization over parallel computer with message passing
- Here: MPI + OpenMP



Matrix vector OpenMP

```
!$OMP PARALLEL DO &
!$OMP & SHARED(Aloc,x,n,nloc)&
!$OMP & PRIVATE(yloc,i,j) &
!$OMP & REDUCTION(+:yloc)
  do j = 1, n
    do i = 1, nloc
      yloc(i)=yloc(i)+Aloc(i,j)*x(j)
    end do
  end do
!$OMP END PARALLEL DO
```

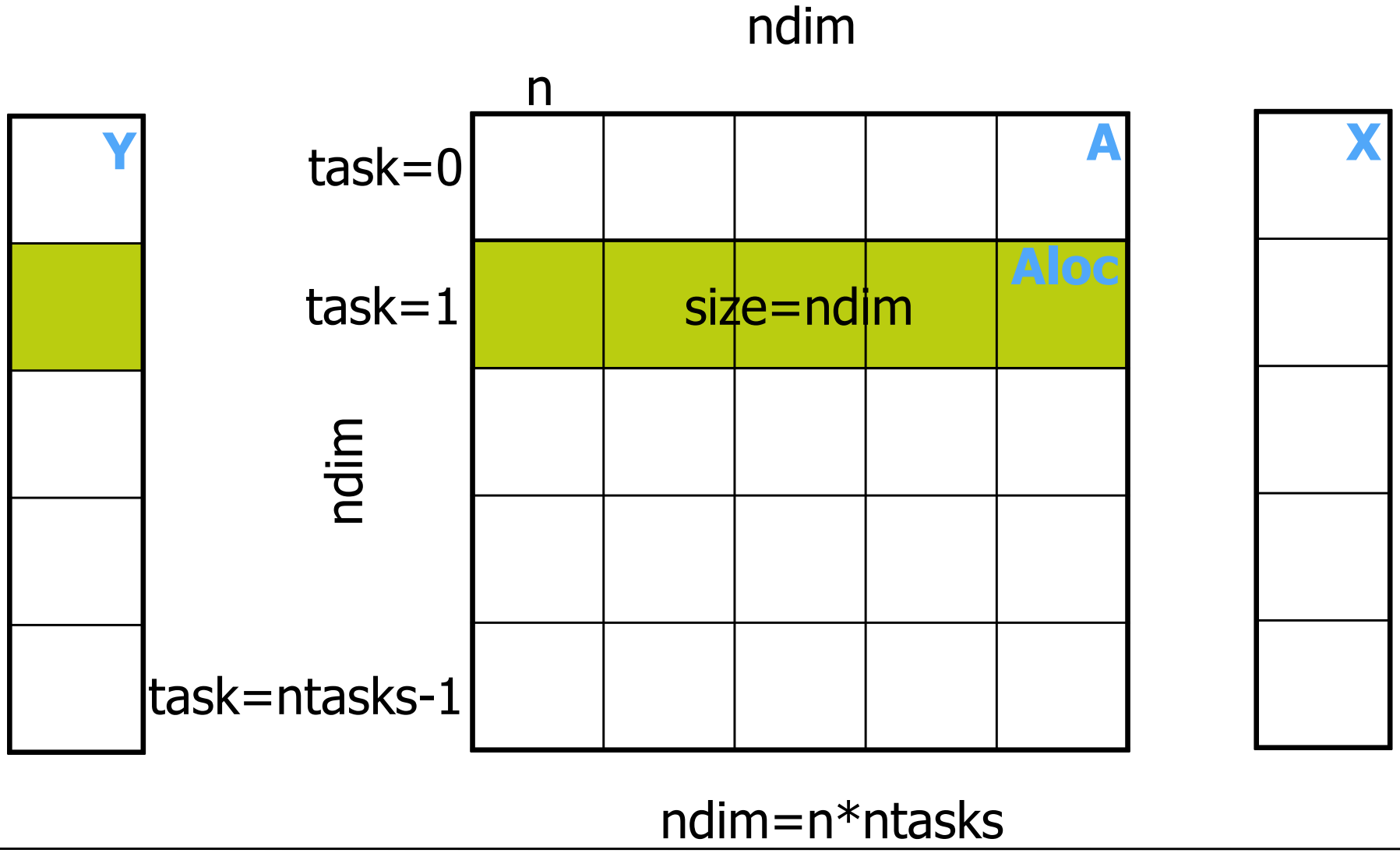
”Fine-grained”

```
#pragma omp parallel for \
  shared(A,y,x) private(i,j,asum) \
  schedule(guided,chunk)
  for (i=0;i<n;i++){
    asum=0.0;
    for (j=0;j<n;j++)
      asum += A[i][j]*x[j];
    y[i]=asum;
  }
```

Exercise: MatrixVector

- From directory MatrixVector
  - inserted MPI calls from previous exercise
  - insert OpenMP directives for local loop (see previous slide)
  - contains 2 OpenMP based solutions
  - check performance running pure MPI and Hybrid, use job.slurm to submit job to queue

Collectives: Matrix-Vector



Thread support in MPI

- MPI standard defines four levels of support
  - MPI_THREAD_SINGLE
    - Only one thread allowed
  - MPI_THREAD_FUNNELED
    - Only master thread allowed to make an MPI call
  - MPI_THREAD_SERIALIZED
    - All threads allowed to make MPI calls, but not concurrently
  - MPI_THREAD_MULTIPLE
    - No restrictions