How PETSc is currently handling multicore
(not GPUs (yet 😊))

Shri Abhyankar
Jed Brown
Barry Smith
Multicore is distributed memory with the ability to directly address “remote local” memory:

- via contiguous addresses and/or
- via separate pointers/arrays
Contiguous addresses may access different physical memory

Or a conceptually single array may not have contiguous addresses (MPI style via message passing or one-sided operations)
Example: Sparse matrix vector product

Contiguous addresses may access different physical memory

Non-contiguous addresses of conceptually single array (MPI)
PETSc User Interface for MPI

Vec/MatCreate(MPI_Comm comm, Vec *avector)

MPI_Comm determines the (subset of all) physical memories the vectors/matrices data is stored on and the (subset of all) physical cores that the vector operations are performed on. Reasonable default of MPI_COMM_WORLD

PETSc User Interface for hybrid MPI + X (OpenMP, Pthreads, TBB, ???, GPU?)

Vec/MatCreate(MPI_Comm comm, Vec *avector)

MPI_Comm attribute (PetscThreadComm) determines (subset of all) memory sockets and (subset of all) cores that “share” the vectors/matrices data on each node of the MPI communicator.

Data is stored in contiguous arrays that lie across the (subset of all) memory sockets. (Shared memory style)
PETSc Implementation

1. Numerical (threaded) kernel code is separated from “control” code in its own routine and then “launched” by controlling code.

2. Threaded kernel code “determines” its data region to work on and then performs computation.

```c
PetscErrorCode VecAXPY(Vec yin, double alpha, Vec xin) {
    MPI_Comm comm = ((PetscObject)yin)->comm;
    PetscScalar *scalar;
    PetscThreadCommGetScalars(comm, &scalar, ...);
    *scalar = alpha;
    PetscThreadCommRunKernel(comm, VecAXPY_kernel, 3, yin, scalar, xin);
    // does not block for completion of operation
}
```
VecAXPY_kernel(int id, Vec yin, double *alpha, Vec xin)
{
    const PetscScalar *xarray;
    PetscScalar       *yarray;
    PetscInt          *trstarts=yin->map->trstarts,start,n;

    start = trstarts[id];
    n     = trstarts[id+1] - start;
    VecGetArrayRead(xin,&xarray);
    VecGetArray(yin,&yarray);
    BLASaxpy_(n,alpha,xarray+start,1,yarray+start,1);
}
PETSc Implementation with Reduction

1. Numerical (threaded) kernel code is separated from “control” code in its own routine and then “launched” by controlling code.
2. Threaded kernel code “determines” its data region to work on and then performs computation.

PetscErrorCode VecDot (Vec yin, Vec xin,double *result)
{
    MPI_Comm comm = ((PetscObject)yin)->comm;
    PetscThreadCommRedCtx red;

    PetscThreadReductionBegin(comm,THREADCOMM_SUM,PETSC_SCALAR,&red);
    PetscThreadCommRunKernel(comm, VecDot_kernel,3,xin,yin,red);
    PetscThreadReductionEnd(red,result); // blocks here for result
}
VecDot_kernel(int id, Vec yin, Vec xin, PetscThreadCommCommRedCtx red)
{
    const PetscScalar *xarray, *yarray;
    int *trstarts = yin->map->trstarts, start, n;
    double dot_loc;
    start = trstarts[id];
    n = trstarts[id+1] - start;
    VecGetArrayRead(xin, &xarray);
    VecGetArrayRead(yin, &yarray);
    dot_loc = BLASdot_(n, xarray+start, 1, yarray+start, 1);
    PetscThreadReductionKernelBegin(id, red, &dot_loc);
}
The only OpenMP code in PETSc

PetscThreadCommRunKernel_OpenMP(MPI_Comm comm, PetscThreadCommJobCtx job)
{
    PetscThreadComm tcomm;
    int id=0;

    PetscCommGetThreadComm(comm,&tcomm);
    #pragma omp parallel num_threads(tcomm->nworkThreads) shared(comm,job)
       private(id)
    {
        id =omp_get_thread_num();
        PetscRunKernel(id,job->nargs,job);
    }
}
Users Computational Code

Use OpenMP directly with PETSc vector (and matrix) data structures or

Use TBB directly with PETSc vector (and matrix) data structures or

Use PETSc’s kernel launcher

Use pthreads for minimum synchronization and the possibility of having multiple different operations running at once.

Use OpenMP or TBB when user code already uses these threading systems and want to share the thread pool with PETSc.

Kernels may be called by other kernels; more potential efficient code reuse then possible with inlined OpenMP kernels)
Performance Issues

- Portions of contiguous arrays stored on “correct” memory socket
  - Each thread “initializes” its part of the array (first touch)
- Pages of contiguous arrays align with memory sockets
  - We manage the “local sizes” of array for each memory socket, don’t just OpenMP default
- Avoid all unneeded locks and synchronization points
  - They are expensive
  - Kernel launch and completion have no atomic instructions (only memory barriers)
- Manage reductions ourselves
- Work stealing?? (Dynamic load balance within a node)
  - Partition arrays on each memory socket into many smaller “portions”.
  - Launch kernels for each “portion” “sequentially” on cores local to that “portions” memory socket.
  - If all the cores for one memory socket “get behind” and other cores are free then launch excessive “portions” of a memory socket onto those “other cores”.
  - Eventually would like “portions” to migrate to the other memory socket, numa_move_pages()?