MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University

Lecture 41:

Parallelization on a cluster of distributed memory machines

Part 1: Introduction to MPI

Shared memory

In the previous lecture:

- There was a single address space
- All parallel threads of execution have access to all data

Advantage:

- Makes parallelization simpler

Disadvantages:

- Problem size limited by
  - number of cores on your machine
  - amount of memory on your machine
  - memory bandwidth
- Need synchronisation via locks
- Makes it too easy to avoid hard decisions

Shared memory

Example:

- Only one Triangulation, DoFHandler, matrix, rhs vector
- Multiple threads work in parallel to
  – assemble linear system
  – perform matrix-vector products
  – estimate the error per cell
  – generate graphical output for each cell
- All threads access the same global objects

For examples, see several of the step-xx programs and the "Parallel computing with multiple processors accessing shared memory" documentation module

Shared vs. distributed memory

This lecture:

- Multiple machines with their own address spaces
- No direct access to remote data
- Data has to be transported explicitly between machines

Advantage:

- (Almost) unlimited number of cores and memory
- Often scales *better* in practice

Disadvantages:

- Much more complicated programming model
- Requires entirely different way of thinking

- Practical difficulties debugging, profiling, ...

Distributed memory

Example:

- One Triangulation, DoFHandler, matrix, rhs vector object per processor
- Union of these objects represent global object
- Multiple programs work in parallel to
  - assemble *their part of the* linear system
  - perform *their part of the* matrix-vector products
  - estimate the error *on their cells*
  - generate graphical output for each *of their cells*
- Each program only accesses their part of global objects

See step-40/32/42 and the “Parallel computing with multiple processors using distributed memory” module

Distributed memory

There are many ways to do distributed memory computing:

- Message passing interface (MPI)
- Remote procedure calls (RPC)
- Partitioned global address space (PGAS) languages:
  – Unified Parallel C (UPC – an extension to C)
  – Coarray Fortran (part of Fortran 2008)
  – Chapel, X10, Titanium

Message Passing Interface (MPI)

MPI's model is simple:

- The "universe" consists of "processes"
- Typically:
  - One single-threaded process per core
  - One multi-threaded process per machine
- Processes can send "messages" to other processes…
- …but nothing happens if the other side is not listening

Mental model: Sending letters through the mail system

Message Passing Interface (MPI)

MPI's model implies:

- You can't "just access" data of another process

- Instead, option 1:
  - you need to send a request message
  - other side has to pick up message
  - other side has to know what to do
  - other side has to send a message with the data
  - you have to pick up message

- Option 2:
  - depending on phase of program, I know when someone else needs my data → send it
  - I will know who sent me data → go get it

Message Passing Interface (MPI)

MPI's model implies:

- You can't “just access” data of another process
- Instead...

This is bothersome to program. However:

- It exposes to the programmer what is happening
- Processes can do other things between sending a message and waiting for the next
- Has been shown to scale to >1M processes

Message Passing Interface (MPI)

MPI implementations:

- MPI is defined as a set of
  - functions
  - data types
  - constants

  with bindings to C and Fortran
- Is not a language on its own
- Can be compiled by a standard C/Fortran compiler
- Is typically compiled using a specific compiler wrapper:

```
mpicc -c myprog.c -o myprog.o
mpiCC -c myprog.cc -o myprog.o
mpif90 -c myprog.f90 -o myprog.o
```

- Bindings to many other languages exist

Message Passing Interface (MPI)

MPI's bottom layer:

- Send messages from one processor to others
- See if there is a message from any/one particular process
- Receive the message

Example (send on process 2 to process 13):

```
double d = foo();
MPI_Send (/*data=*/&d, /*count=*/1, /*type=*/MPI_DOUBLE,
          /*dest=*/13, /*tag=*/42,
          /*universe=*/MPI_COMM_WORLD);
```

Message Passing Interface (MPI)

MPI's bottom layer:

- Send messages from one processor to others
- See if there is a message from any/one particular process
- Receive the message

Example (query for data from process 13):

```
MPI_Status status;
int                message_available;
MPI_Iprobe (/*source=*/13, /*tag=*/42,
            /*yesno=*/message_available,
            /*universe=*/MPI_COMM_WORLD,
            /*status=*/&status);
```

Note: One can also specify “anywhere”/”any tag”.

Message Passing Interface (MPI)

MPI's bottom layer:

- Send messages from one processor to others
- See if there is a message from any/one particular process
- Receive the message

Example (receive on process 13):

```
double d;
MPI_Status status;
MPI_Recv (/*data=*/&d, /*count=*/1, /*type=*/MPI_DOUBLE,
          /*source=*/2, /*tag=*/42,
          /*universe=*/MPI_COMM_WORLD,
          /*status=*/&status);
```

Note: One can also specify “anywhere”/”any tag”.

Message Passing Interface (MPI)

MPI's bottom layer:

- Send messages from one processor to others
- See if there is a message from any/one particular process
- Receive the message

Notes:

- *MPI_Send* blocks the program: function only returns when the data is out the door
- *MPI_Recv* blocks the program: function only returns when
  – a message has come in
  – the data is in the final location
- There are also non-blocking start/end versions (*MPI_Isend*, *MPI_Irecv*, *MPI_Wait*)

Message Passing Interface (MPI)

MPI's higher layers: Collective operations

- Internally implemented by sending messages
- Available operations:
  - Barrier
  - Broadcast (one item from one to all)
  - Scatter (many items from one to all),
  - Gather (from all to one), AllGather (all to all)
  - Reduce (e.g. sum from all), AllReduce

Note: Collective operations lead to deadlocks if some processes do not participate!

Message Passing Interface (MPI)

Example: Barrier use for timing (pseudocode)

```
… do something …
MPI_Barrier (MPI_COMM_WORLD);

std::time_point start = std::now();          // get current time
foo();                                       // may contain MPI calls
std::time_point end_local = std::now();      // get current time

MPI_Barrier (MPI_COMM_WORLD);
std::time_point end_global = std::now();     // get current time

std::duration local_time   = end_local – start;
std::duration global_time = end_global – start;
```

Note: Different processes will compute different values.

Message Passing Interface (MPI)

Example: Reduction

```
parallel::distributed::Triangulation<dim>  triangulation;
… create triangulation …

unsigned int my_cells = triangulation.n_locally_owned_cells();
unsigned int global_cells;

MPI_Reduce (&my_cells, &global_cells, MPI_UNSIGNED, 1,
            /*operation=*/MPI_SUM,
            /*root=*/0,
            MPI_COMM_WORLD);
```

Note 1: Only the root (processor) gets the result.

Note 2: Implemented by (i) everyone sending the root a message, or (ii) hierarchical reduction on a tree

Message Passing Interface (MPI)

Example: AllReduce

```
parallel::distributed::Triangulation<dim>  triangulation;
… create triangulation …

unsigned int my_cells = triangulation.n_locally_owned_cells();
unsigned int global_cells;

MPI_Allreduce (&my_cells, &global_cells, MPI_UNSIGNED, 1,
               /*operation=*/MPI_SUM,
               MPI_COMM_WORLD);
```

Note 1: All processors now get the result.

Note 2: Can be implemented by MPI_Reduce + MPI_Broadcast

Message Passing Interface (MPI)

MPI's higher layers: Communicators

- MPI_COMM_WORLD denotes the “universe” of all MPI processes
- Corresponds to a “mail service” (a communicator)
- Addresses are the “ranks” of each process in a communicator

- One can form subsets of a communicator
- Forms the basis for collective operations among a subset of processes
- Useful if subsets of processors do different tasks

Message Passing Interface (MPI)

MPI's higher layers: I/O

- Fact: There is a bottleneck if 1,000 machines write to the file system at the same time

- MPI provides ways to make this more efficient

Message Passing Interface (MPI)

Also in MPI:

- "One-sided communication": directly writing into and reading from another process's memory space
- Topologies: mapping network characteristics to MPI
- Starting additional MPI processes

More information on MPI:

http://www.mpi-forum.org/

An MPI example: MatVec

Situation:

- Multiply a large *NxN* matrix by a vector of size *N*
- Matrix is assumed to be dense

- Every one of *P* processors stores *N/P* rows of the matrix
- Every processor stores *N/P* elements of each vector

- For simplicity: *N* is a multiple of *P*

An MPI example: MatVec

```
struct ParallelVector {
    unsigned int size;
    unsigned int my_elements_begin;
    unsigned int my_elements_end;
    double *elements;

    ParallelVector (unsigned int sz,MPI_Comm comm) {
        size = sz;
        int comm_size, my_rank;
        MPI_Comm_size (comm, &comm_size);
        MPI_Comm_rank (comm, &my_rank);
        my_elements_begin = size/comm_size*my_rank;
        my_elements_end = size/comm_size*(my_rank+1);
        elements = new double[my_elements_end-my_elements_begin];
    }
};
```

An MPI example: MatVec

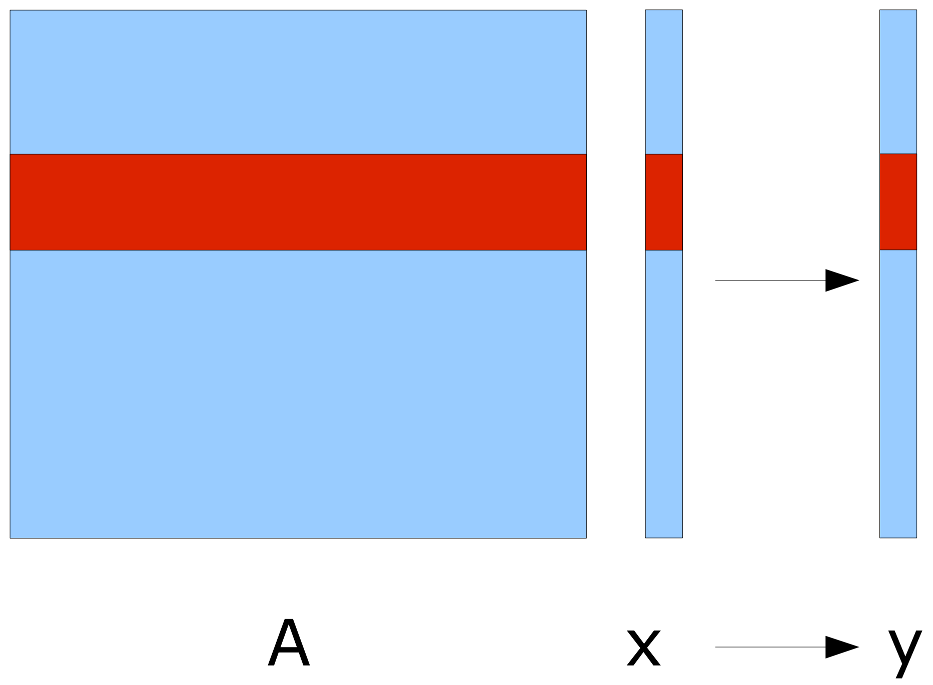
```
struct ParallelSquareMatrix {
    unsigned int size;
    unsigned int my_rows_begin;
    unsigned int my_rows_end;
    double *elements;

    ParallelSquareMatrix (unsigned int sz,MPI_Comm comm) {
        size = sz;
        int comm_size, my_rank;
        MPI_Comm_size (comm, &comm_size);
        MPI_Comm_rank (comm, &my_rank);
        my_rows_begin = size/comm_size*my_rank;
        my_rows_end = size/comm_size*(my_rank+1);
        elements = new double[(my_rows_end-my_rows_begin)*size];
    }
};
```

An MPI example: MatVec

What does processor *P* need:

- Graphical representation of what *P* owns:



- To compute the *locally owned* elements of *y*, processor *P* needs **all** elements of *x*

An MPI example: MatVec

```
void vmult (A, x, y) {
    int comm_size=..., my_rank=...;
    for (row_block=0; row_block<comm_size; ++row_block)
       if (row_block == my_rank) {
          for (col_block=0; col_block<comm_size; ++col_block)
             if (col_block == my_rank) {
                for (i=A.my_rows_begin; i<A.my_rows_end; ++i)
                  for (j=A.size/comm_size*col_block; ...)
                     y.elements[i-y.my_rows_begin] = A[...i,j...] * x[...j...];
             } else {
                double *tmp = new double[A.size/comm_size];
                MPI_Recv (tmp, …, row_block, …);
                for (i=A.my_rows_begin; i<A.my_rows_end; ++i)
                  for (j=A.size/comm_size*col_block; ...)
                     y.elements[i-y.my_rows_begin] = A[...i,j...] * tmp[...j...];
                delete tmp;
             }
       } else {
          MPI_Send (x.elements, …, row_block, …);
       }
}
```

An MPI example: MatVec

Analysis of this algorithm

- We only send data right when we need it:
  – receiving processor has to wait
  – has nothing to do in the meantime
  A better algorithm would:
  – send out its data to all other processors
  – receive messages as needed (maybe already here)
- As a general rule:
  – send data as soon as possible
  – receive it as late as possible
  – try to interleave computations between sends/receives
- We repeatedly allocate/deallocate memory – should set up buffer only once

An MPI example: MatVec

```
void vmult (A, x, y) {
    int comm_size=..., my_rank=...;
    for (row_block=0; row_block<comm_size; ++row_block)
       if (row_block != my_rank)
          MPI_Send (x.elements, …, row_block, …);

   col_block = my_rank;
   for (i=A.my_rows_begin; i<A.my_rows_end; ++i)
     for (j=A.size/comm_size*col_block; ...)
       y.elements[i-y.my_rows_begin] = A[...i,j...] * x[...j...];

   double *tmp = new double[A.size/comm_size];
    for (col_block=0; col_block<comm_size; ++col_block)
       if (col_block != my_rank) {
          MPI_Recv (tmp, …, row_block, …);
          for (i=A.my_rows_begin; i<A.my_rows_end; ++i)
              for (j=A.size/comm_size*col_block; ...)
                  y.elements[i-y.my_rows_begin] = A[...i,j...] * tmp[...j...];
       }
    delete tmp;
}
```

Message Passing Interface (MPI)

Notes on using MPI:

- Usually, algorithms need data that resides elsewhere
- Communication needed

- Distributed computing lives in the conflict zone between
  - trying to keep as much data available locally to avoid communication
  - not creating a memory/CPU bottleneck

- MPI makes the flow of information explicit
- Forces programmer to design data structures/algorithms for communication

- Typical programs have relatively few MPI calls

Message Passing Interface (MPI)

Alternatives to MPI:

- boost::mpi is nice, but doesn't buy much in practice

- Partitioned Global Address Space (PGAS) languages like Co-Array Fortran, UPC, Chapel, X10, …:

  Pros:
  – offer nicer syntax
  – communication is part of the language
  Cons:
  – typically no concept of "communicators"
  – communication is implicit
  – encourages poor data structure/algorithm design

MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University