High Performance Computing

Autumn, 2018

Lecture 18

Imperial College
London

Prasun Ray

4 December 2018

Notes

M3C final project: please keep an eye on *online* assignment for clarifications/ corrections

Feedback: All provided marks are *provisional* and subject to rescaling by an exam committee in June

Check mpif90/mpiexec installation, both need to be in path, see notes on project

MPI datatypes

List of MPI datatypes (Fortran)

MPI Datatype	Fortran Datatype
MPI_CHARACTER	CHARACTER
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_INTEGER	INTEGER
MPI_LOGICAL	LOGICAL
MPI_PACKED	
MPI_BYTE	

MPI reductions

List of MPI reductions:

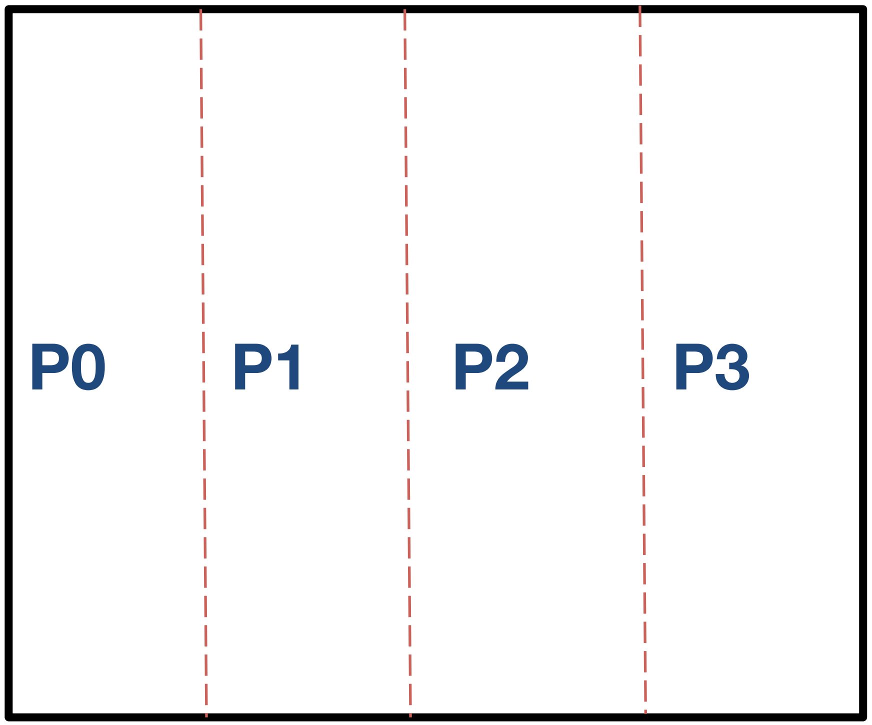
Reduction	Description	Datatype(s)
MPI_MAX	maximum	integer, floating point
MPI_MIN	minimum	
MPI_SUM	sum	integer, floating point, complex, multilanguage types
MPI_PROD	product	
MPI_LAND	logical and	logical
MPI_LOR	logical or	
MPI_LXOR	logical xor	
MPI_BAND	bitwise and	integer, byte, multilanguage types
MPI_BOR	bitwise or	
MPI_BXOR	bitwise xor	
MPI_MAXLOC	max value and location	MPI_DOUBLE_INT and such
MPI_MINLOC	min value and location	

Special datatypes needed for maxloc,minloc, see:
https://www.open-mpi.org/doc/v2.0/man3/MPI_Reduce.3.php

2-D (steady) heat equation

What is best domain decomposition?

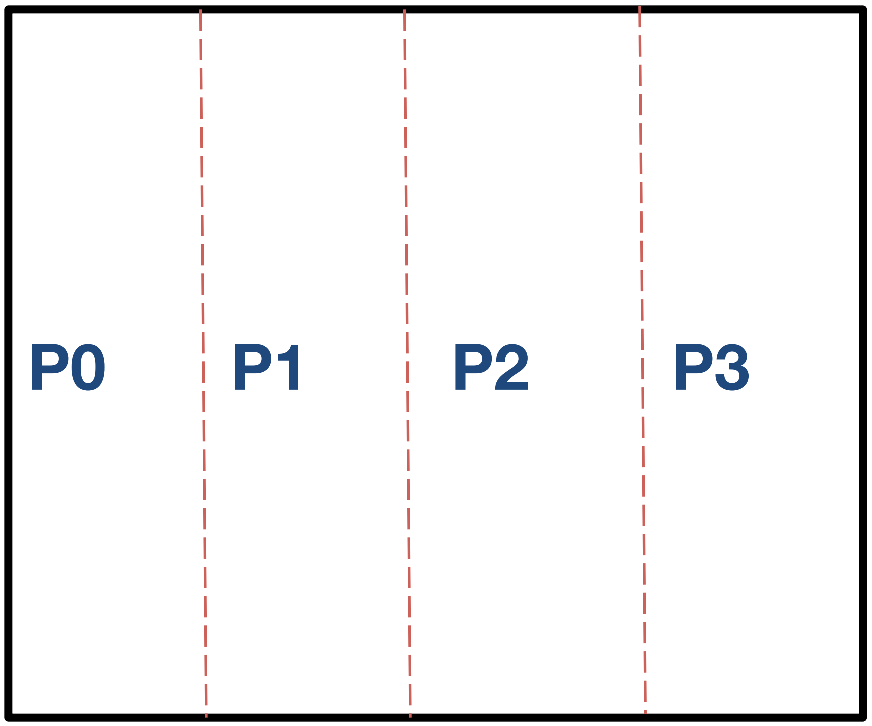
If we have four processors, can try:



2-D (steady) heat equation

What is best domain decomposition?

If we have four processors, can try:



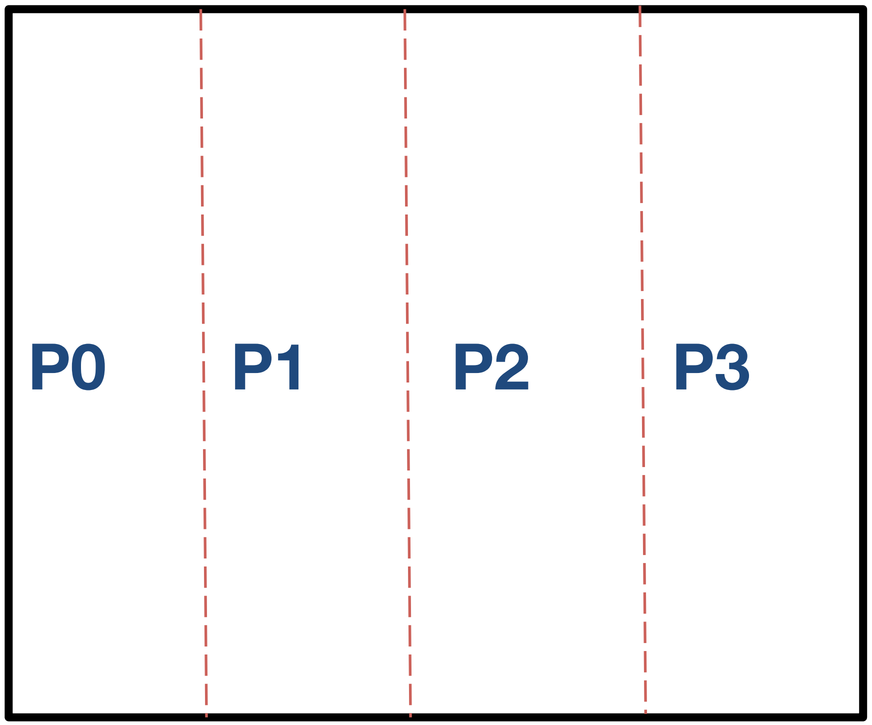
Then, parallelization is essentially same as differentiation example:

- **Loop across rows**
- **At “boundaries”, send/recv data needed to compute second derivative**
- **Reduce max(|deltaT|)**

2-D (steady) heat equation

What is best domain decomposition?

If we have four processors, can try:

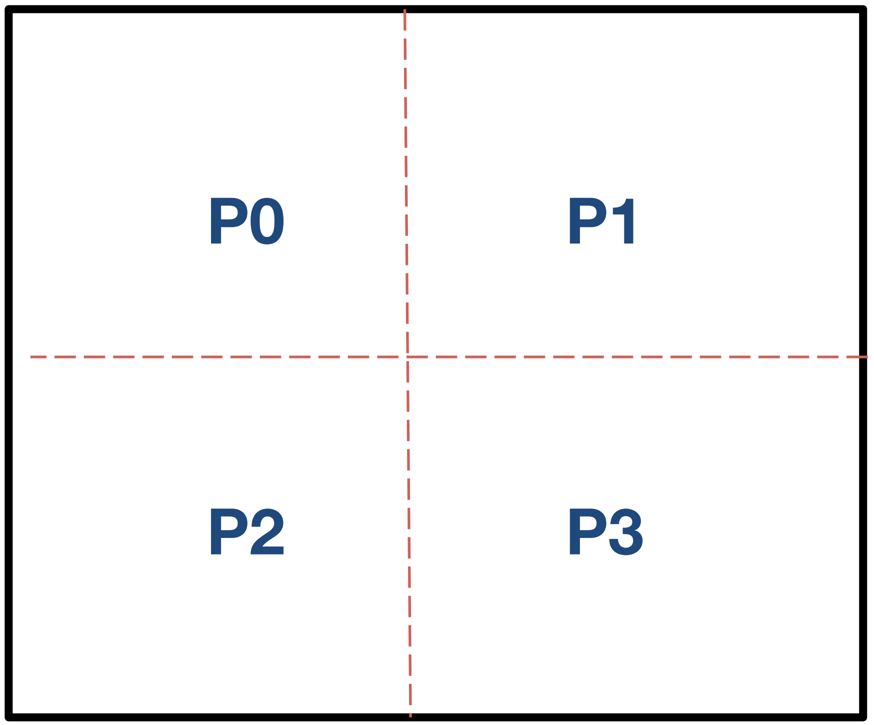


- **But is the 1D decomposition the best?**
- **Want to minimize communication**
- **M "layers": of a n x n grid:**
  - **(M-1)*n boundary points**

2-D (steady) heat equation

What is best domain decomposition?

If we have four processors, can also try:

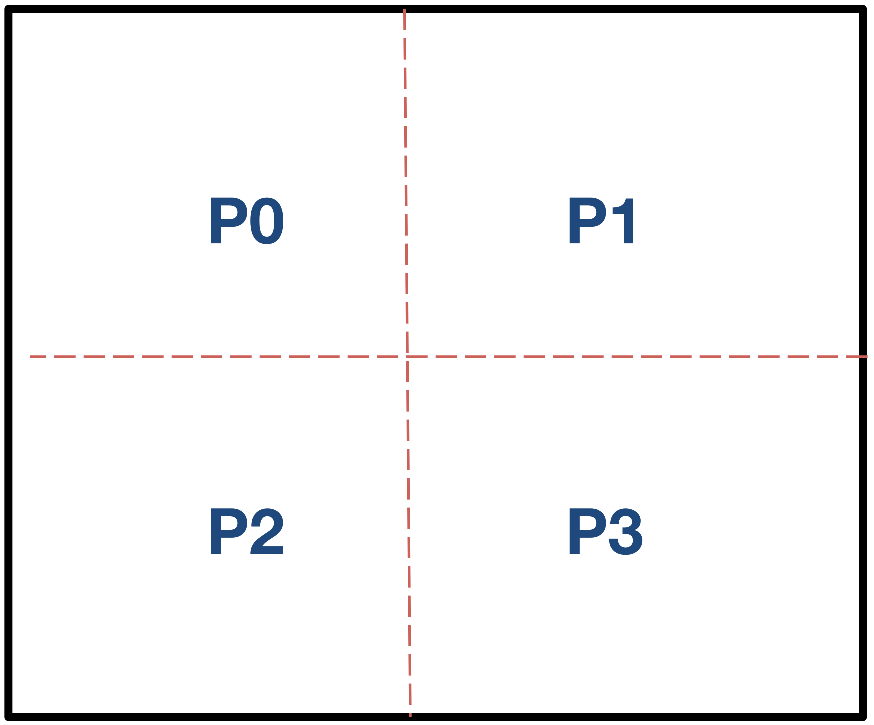


- **But is the 1D decomposition the best?**

2-D (steady) heat equation

What is best domain decomposition?

If we have four processors, can also try:

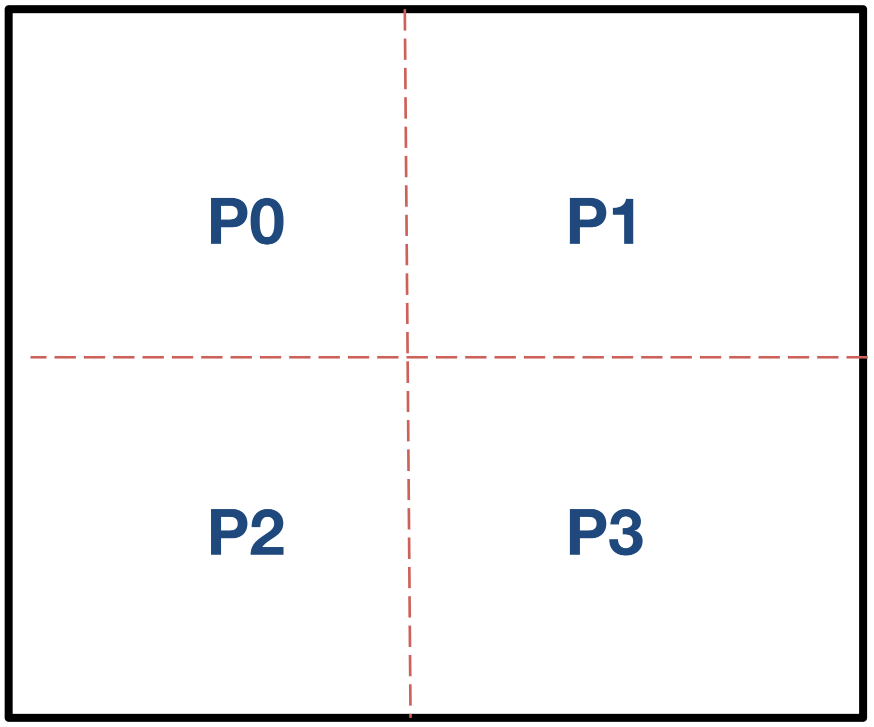


- But is the 1D decomposition the best?
- Want to minimize communication
- M "boxes": of a n x n grid:
    - Each interior box has 2n/sqrt(M) boundary points
    - Total: 2n*(sqrt(M)-1) boundary points

2-D (steady) heat equation

What is best domain decomposition?

If we have four processors, can also try:



"Boundary points" are on the red dashed lines

- **But is the 1D decomposition the best?**
- **Want to minimize communication**
- **M "boxes": of a n x n grid:**
  - **Each box has 2n/sqrt(M) boundary points**
  - **Total: 2n*(sqrt(M) -1) boundary points**
- **Boxes: less communication, but more difficult to implement!**

2-D (steady) heat equation

- **MPI provides tools for creating and managing complex "topologies"**

- **For example, to create a 4 x 3 "grid" of processes:**

```
    call MPI_cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder,
                                                  new_comm, ierr)
```

with: ndims = 2, dims = (/4,3/), periods = (/.false.,/.true/), reorder = .false.

- **Here,** periods **sets periodic boundary conditions along the three columns**

2-D (steady) heat equation

- **MPI provides tools for creating and managing complex "topologies"**

- **For example, to create a 4 x 3 "grid" of processes:**

```
call MPI_cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder,
                                                      new_comm, ierr)
```

with: ndims = 2, dims = (/4,3/), periods = (/.false.,/.true/), reorder = .false.

- **Here,** periods **sets periodic boundary conditions along the three columns**

- **Other useful commands with the new communicator,** new_comm**:**
  - MPI_Cart_coords**: given process id, provides (i,j) coordinate**

  - MPI_Cart_rank**: given (i,j), provides id (0, 1, 2, …, numprocs)**

- **Most useful:** MPI_Cart_shift**: provides id of neighboring processes in horizontal or vertical direction**
  - **Use to set up send/recv sequences needed for exchanging boundary data.**

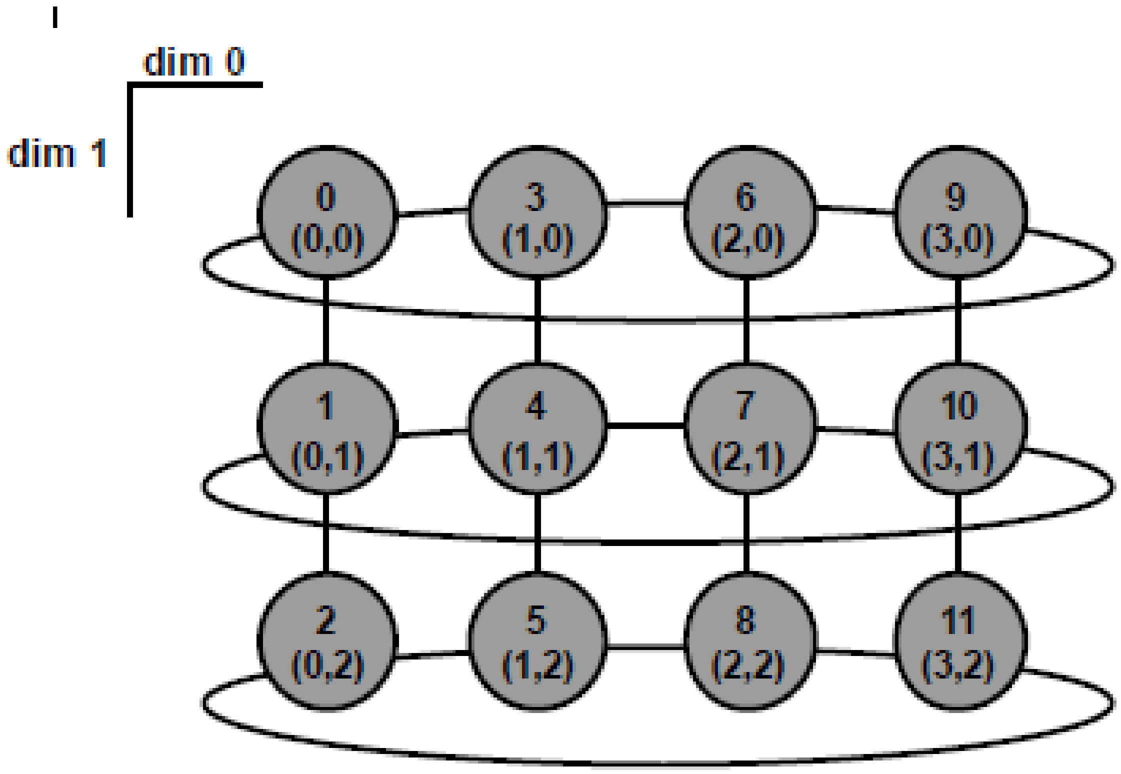
2-D (steady) heat equation

- **Most useful:** MPI_Cart_shift**: provides id of neighboring processes in horizontal or vertical direction**
  - **Use to set up send/recv sequences needed for exchanging boundary data.**

- **How to decide on process grid dimensions?**

- MPI_Dims_create**:**
Given number number of processes and dimensions, outputs process grid Dimensions (4,3 in picture →)

- **e.g. 400 x 300 grid points:**
4 x 3 process grid with 100 x 100 points on each grid



Synchronization (OpenMP)

- **Some threads may be given more work than others**
- **One thread may complete its tasks quickly and move very far ahead of the other threads**
- ***Barriers* keep the threads synchronized:**

```
!$OMP parallel

!Some code

!$OMP barrier

!$OMP end parallel
```

- **Threads will not continue past the barrier until all threads reach the barrier**

Synchronization (MPI)

- Same idea as in OpenMP
- Use *MPI_BARRIER* to keep processes synchronized
- *Barriers* keep the threads synchronized:

```
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
```

See *gradient_p.f90* from lecture 16

Timing code

1. **Lazy approach:**

```
$ time mpiexec -n 2 midpointpt

real   0m0.073s
user   0m0.081s
sys 0m0.030s
```

Timing code

1. **Lazy approach:**

```
$ time mpiexec -n 2 midpointpt

real   0m0.073s
user   0m0.081s
sys 0m0.030s
```

2. **Use *MPI_WTIME* to time particular parts of code:**

```
starttime = MPI_WTIME() !***START TIMER***
    !code...
    !
    !
endtime = MPI_WTIME() !***STOP TIMER***
print *, 'time= ',endtime - starttime, 'seconds'
```

Timing code

1. **Lazy approach:**

```
$ time mpiexec -n 2 midpointpt

real   0m0.073s
user   0m0.081s
sys 0m0.030s
```

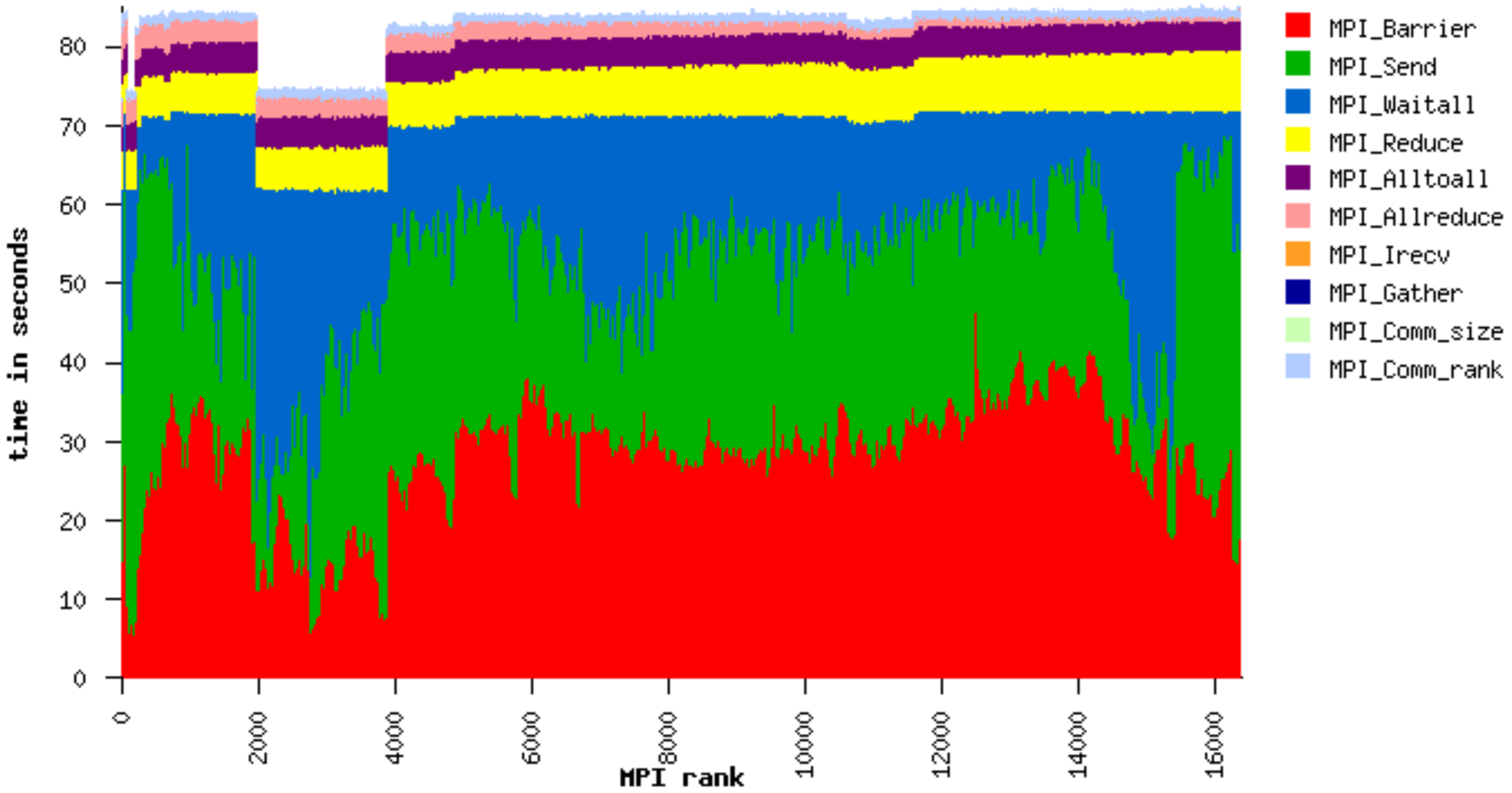
2. **Use *MPI_WTIME* to time particular parts of code:**

```
starttime = MPI_WTIME() !***START TIMER***
    !code...
    !
    !
endtime = MPI_WTIME() !***STOP TIMER***
print *, 'time= ',endtime - starttime, 'seconds'
```

3. **But to get detailed information, use a profiler: VampirTrace, IPM, …**

Timing code

From IPM webpage (*http://ipm-hpc.sourceforge.net/*)**:**



OpenMP vs MPI

OpenMP:

- Advantages:
  - Easy to implement (particularly for loops)
  - Single code for serial and parallel execution
  - Included with most compilers

- Disadvantages:
  - Can only be used for shared-memory computers

OpenMP vs MPI

OpenMP:

- **Advantages:**
  - **Easy to implement (particularly for loops)**
  - **Single code for serial and parallel execution**
  - **Included with most compilers**

- **Disadvantages:**
  - **Can only be used for shared-memory computers**

MPI:

- **Advantages:**
  - **Can be used for distributed- or shared-memory systems**
  - **Distributed memory computing:**
    - **Each process has its own local variables**
    - **Programmer has detailed control over communication**

- **Disadvantages:**
  - **Relatively difficult to implement**
  - **Typically have separate serial and parallel codes**

Choosing the right tools

- We have studied two programming languages and two approaches to parallelization

- When should we use Python, Fortran, Fortran+OpenMP, Fortran+MPI?

Choosing the right tools

- We have studied two programming languages and two approaches to parallelization
- When should we use Python, Fortran, Fortran+OpenMP, Fortran+MPI?
- Rules of thumb for solving PDEs (highly subjective)
    - 1D problems: interpreted language
    - 2D, single equation: compiled language
    - 2D, system of PDEs: compiled language, OpenMP (2-16 cores)
    - 3D, single equation: compiled language, OpenMP or maybe MPI (depending on problem size) (4-128 cores)
    - 3D system of PDES: compiled language + MPI (128+ cores)

Choosing the right tools

- We have studied two programming languages and two approaches to parallelization
- When should we use Python, Fortran, Fortran+OpenMP, Fortran+MPI?
- Rules of thumb for solving PDEs (highly subjective)
  - 1D problems: interpreted language
  - 2D, single equation: compiled language
  - 2D, system of PDEs: compiled language, OpenMP (2-16 cores)
  - 3D, single equation: compiled language, OpenMP or maybe MPI (depending on problem size) (4-128 cores)
  - 3D system of PDES: compiled language + MPI (128+ cores)

But remember: Matlab and Python have libraries which are essentially compiled Fortran/c codes!

Beyond M3C

- We have focused on a few problems
  - evolutionary games, neural networks, diffusion
- But ideas on parallelization can be easily generalized
- Consider extremely large datasets collected by Google (or the NSA!)
- This data *must* be processed with distributed parallel computing
- Want to find patterns, correlations, extrema
- But calculations will require data stored on different machines
  - Google developed *MapReduce* ~15 years ago for these problems

    http://research.google.com/archive/mapreduce.html

    (Google is now using something called, *Cloud Dataflow)*

Simple MapReduce example

Example: Count word occurrences

```
subroutine map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
      EmitIntermediate(w, "1");
```

Subroutine *map:* From document (input_key), read in data (input_value) count occurrences of a word

Simple MapReduce example

Example: Count word occurrences

```
subroutine map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
      EmitIntermediate(w, "1");

subroutine reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
      result += ParseInt(v);
    Emit(AsString(result));
```

Subroutine *map:* From document (input_key), read in data (input_value) count occurrences of a word

Subroutine *reduce:* Sum total number of occurrences computed by *map*

Simple MapReduce example

Example: Count word occurrences

```
subroutine map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
      EmitIntermediate(w, "1");

subroutine reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
      result += ParseInt(v);
    Emit(AsString(result));
```

Similar to midpoint_mpi.f90!:
***map*: each process assigned a subdomain, computes partial sum**

***reduce*: total sum from partial sums**

Subroutine *map:* From document (input_key), read in data (input_value) count occurrences of a word

Subroutine *reduce:* Sum total number of occurrences computed by *map*