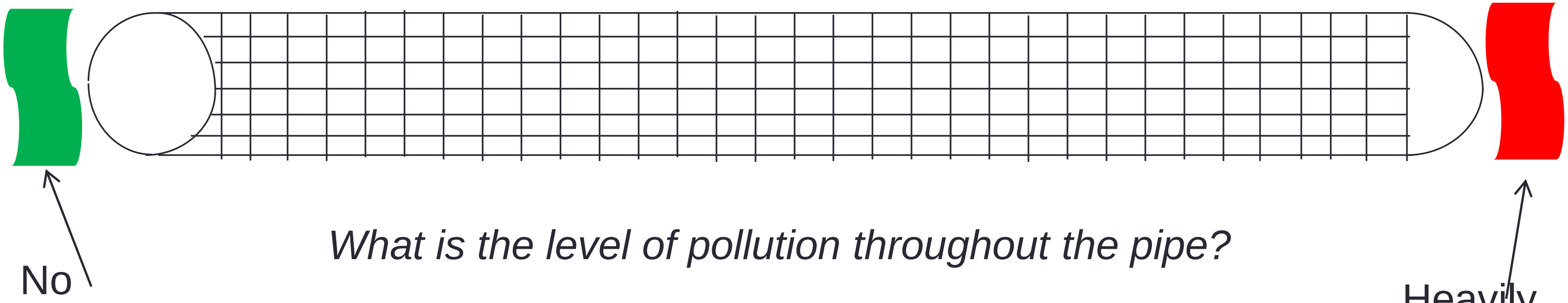
Parallel design patterns ARCHER course

Practical one: Pollution in a pipe

and finding concurrency

Reusing this material

Pollution in a pipe



No pollution

Heavily polluted

What is the level of pollution throughout the pipe?

- Jacobi iteration solving Laplace’s equation for diffusion in 2 dimensions

$$\nabla^2 u = 0$$

```
  for all grid points
     u_new(I,j) = 1/4*(u(i-1, j) + u(i+1, j)
                         + u(i,j+1) + u(i,j-1))
```

- Works in iterations, solving to a specific residual (accuracy)
  - As we parallelise this, the overall number of iterations and residual should be the same as the serial code which is a nice check

Overview of serial code

```
double * u_k = malloc(sizeof(double) * (ny+2) * (ny+2)), * u_kp1 = malloc(sizeof(double) * (nx+2) * (ny+2)), *temp;
initialise(u_k, u_kp1);
double rnorm=0.0, bnorm=0.0, norm;
int i, j, k;
for (i=1;i<=nx;i++) {
    for (j=1;j<=ny;j++) {
        bnorm=bnorm+……….
    }
}
bnorm=sqrt(bnorm);
for (k=0;k<MAX_ITERATIONS;k++) {
    for (i=1;i<=nx;i++) {
        for (j=1;j<=ny;j++) {
            rnorm=rnorm+……….
        }
    }
    norm=sqrt(rnorm)/bnorm;
    if (norm < CONVERGENCE_ACCURACY) break;
    for (i=1;i<=nx;i++) {
        for (j=1;j<=ny;j++) {
            u_kp1[i]=0.25 * ……..
        }
    }
    temp=u_kp1; u_kp1=u_k; u_k=temp;
    rnorm=0.0;
}
```

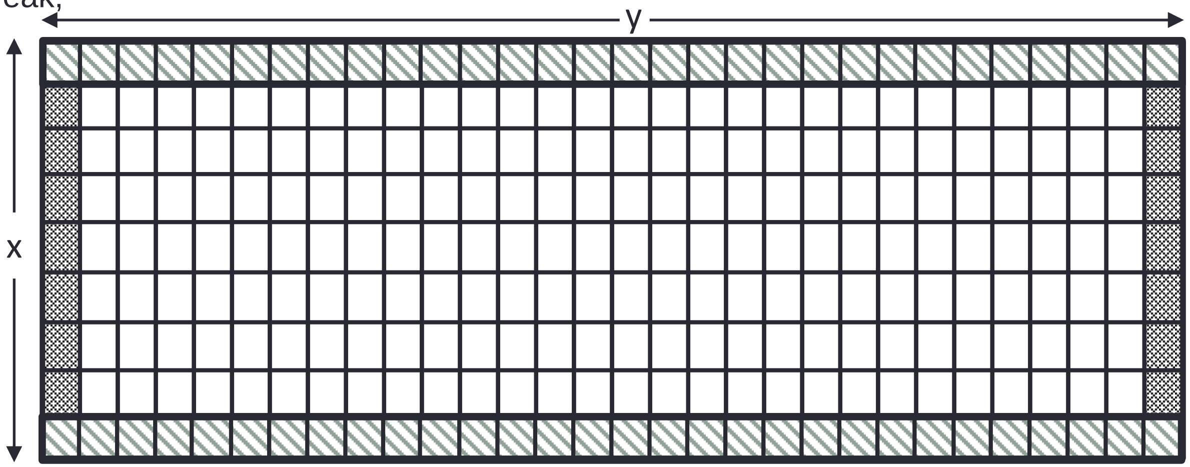
Sets the pollution values at each end of the pipe and the rest to be zero (initial guess)

Compute the initial absolute residual

Compute the absolute residual of the current solution, then divide this by bnorm to get the relative residual (how far we have progressed)

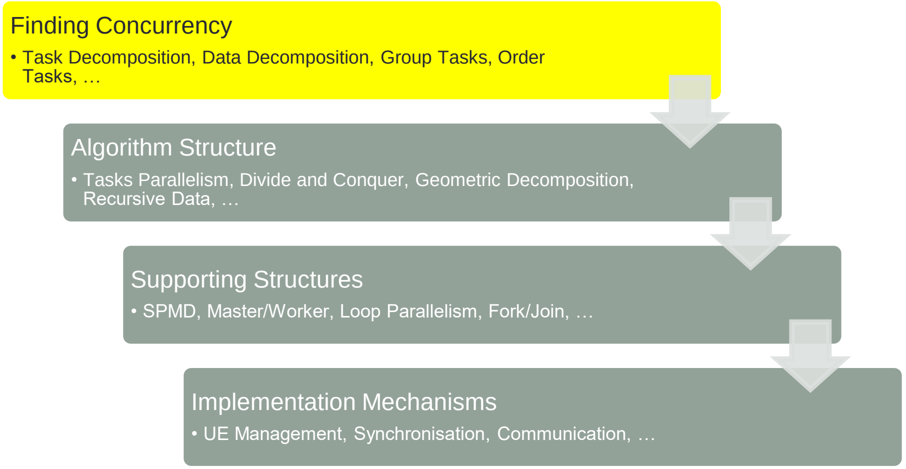
Termination criteria (level of accuracy met)

Jacobi iteration to progress the solution



Finding concurrency

- Split the problem up based upon its functionality
  - Define the tasks and data decomposition implied by these
  - Sometimes independent tasks are easily identified
    - Calls to a function
    - Independent iteration of a loop
    - A number of independent activities being performed
- Driven by data
  - i.e. splitting up of an array into lots of different tasks
- Driven by functionality
  - i.e. distinct tasks, *cat datafile* | *grep "energy"* | *awk '{print $2, $3}'*



Finding Concurrency
- Task Decomposition, Data Decomposition, Group Tasks, Order Tasks, …

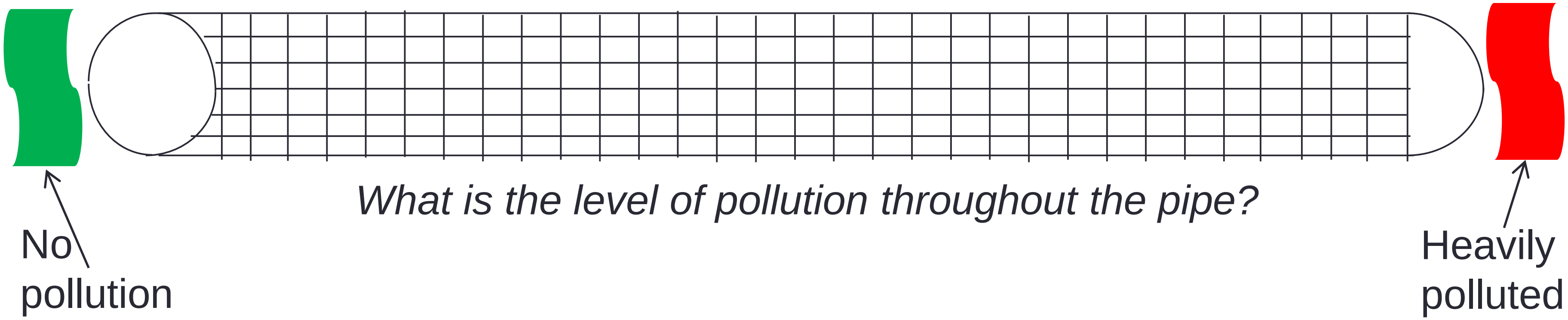
Algorithm Structure
- Tasks Parallelism, Divide and Conquer, Geometric Decomposition, Recursive Data, …

Supporting Structures
- SPMD, Master/Worker, Loop Parallelism, Fork/Join, …

Implementation Mechanisms
- UE Management, Synchronisation, Communication, …

Finding pieces to execute concurrently

- Split the problem up based upon the data it is operating on
  - If it is difficult to split the problem into distinct tasks then instead concentrate on the data is manipulates - especially if this is the most computationally intensive part.
  - E.g. **Arrays:** Concurrency can be defined in terms of updates to different segments of an array which might be decomposed in a variety of different ways.



What is the level of pollution throughout the pipe?

nx+2 * ny+2 tasks, the major organising principal is the data

Ordering the tasks

- Find and account for dependencies resulting from the constrains on the order of execution of tasks
  - Needs to be restrictive enough to satisfy all constraints but no more.

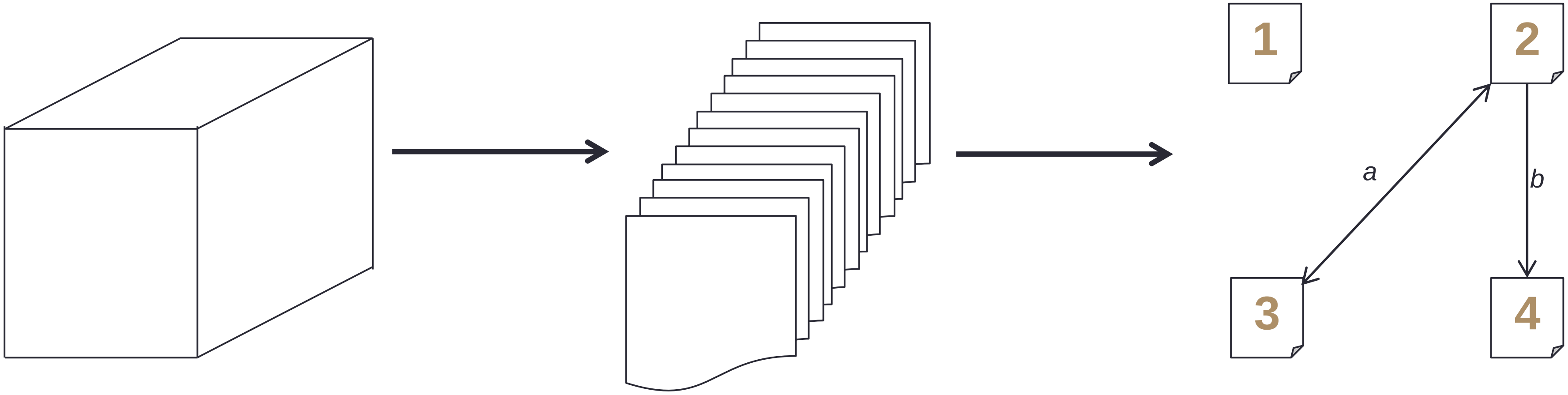
1. Look at the data required by tasks before they can execute
   - Find the task(s) that creates this to form a constraint
2. Can external services impose constraints
   - For instance if a program must write to a file in a specific order
3. Note when an ordering does not exist
   - This is equally important as if tasks can execute independently then there is an opportunity for increased parallelism.

Data sharing between the tasks

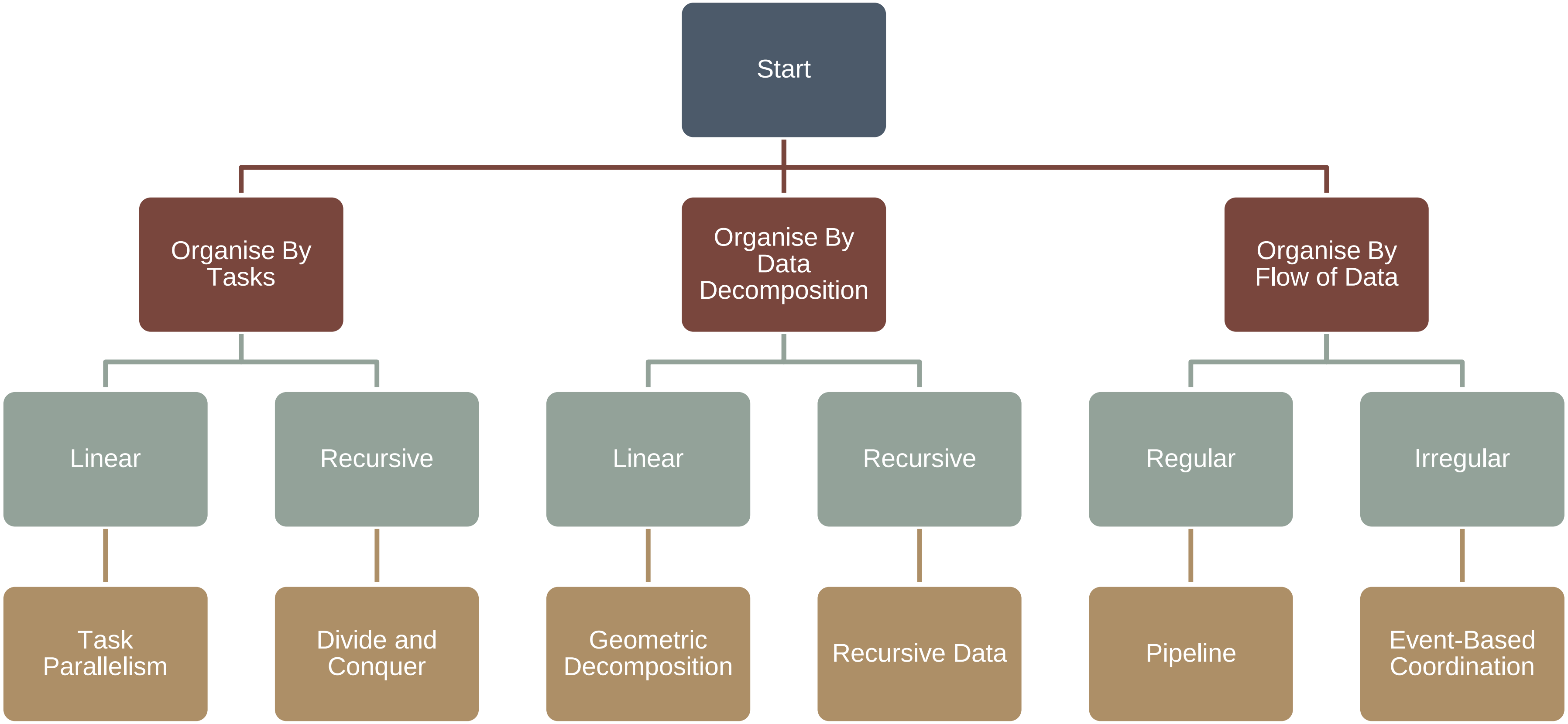
- Important to distinctly identify task local and shared data
  - Tasks might define some global data that must be shared
  - Some tasks might need access to a portion of another task's data
  - How we deal with shared data impacts the correctness (whether it produces the correct result) and performance (not waiting excessively in synchronisation calls and/or reducing communication overhead.)

- Broadly falls into categories of
  - *Read only*: Because there is no modification no protection is needed
  - *Effectively local*: Partitioned into subsets, each accessed by 1 task
  - *Read-write*: The general case and most difficult to deal with
  - *Accumulate*: Updated by many tasks with some operation (eg. Sum)

Design Evaluation



- Now we have our abstract tasks we need to evaluate these
  - You can think of the steps so far as refining the problem to guide your work in the next stage.
  - But are these tasks good enough to move onto the next overall strategy (Algorithm strategy)? Will they give us enough information to work with?



Start

Organise By Tasks

Organise By Data Decomposition

Organise By Flow of Data

Linear

Recursive

Linear

Recursive

Regular

Irregular

Task Parallelism

Divide and Conquer

Geometric Decomposition

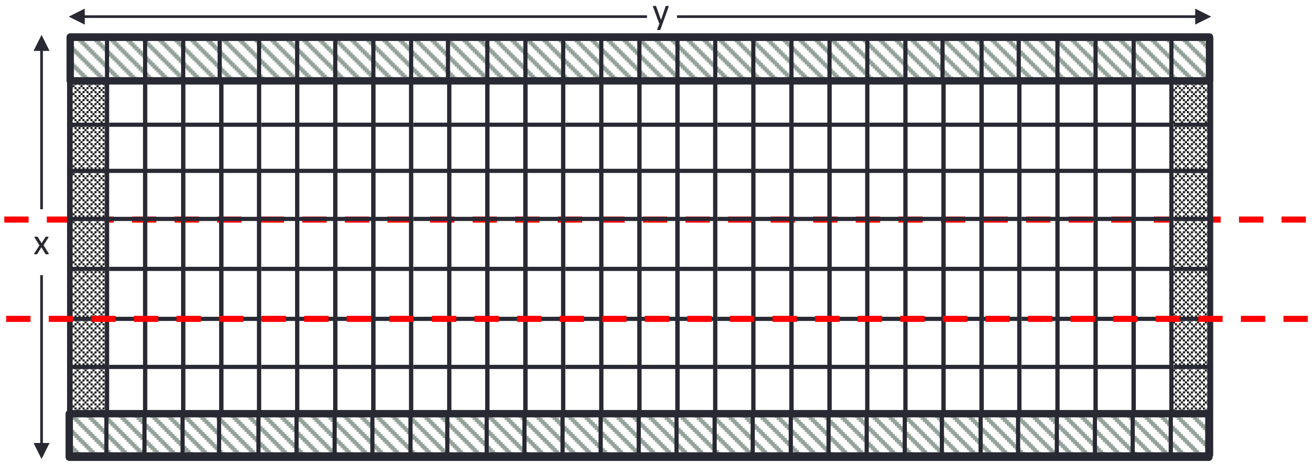
Recursive Data

Pipeline

Event-Based Coordination

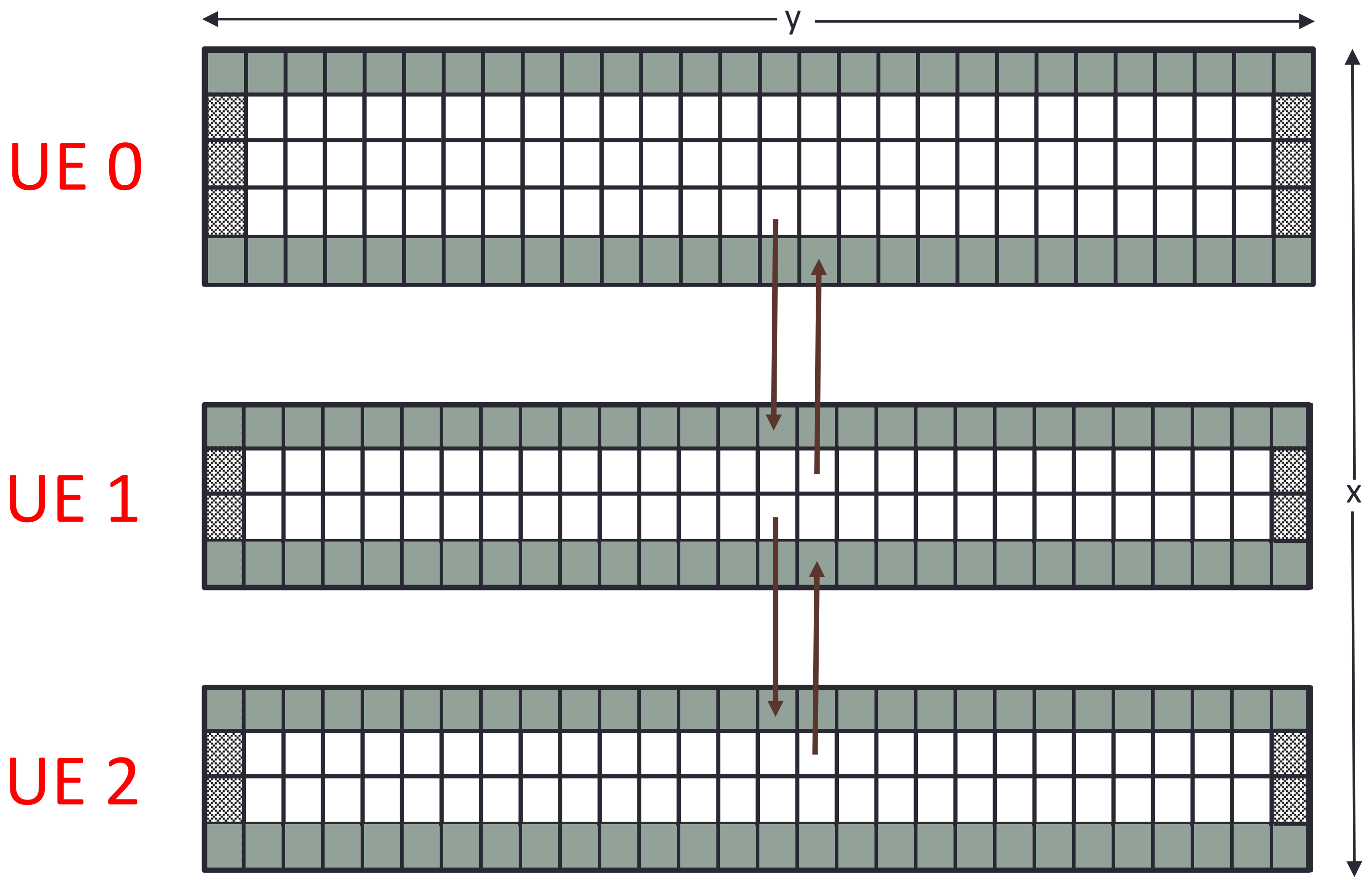
Your task

- Parallelise it!
  - In 1D using geometric decomposition
  - Start with the simplest approach to halo swapping and then add in extra complexity to optimise this



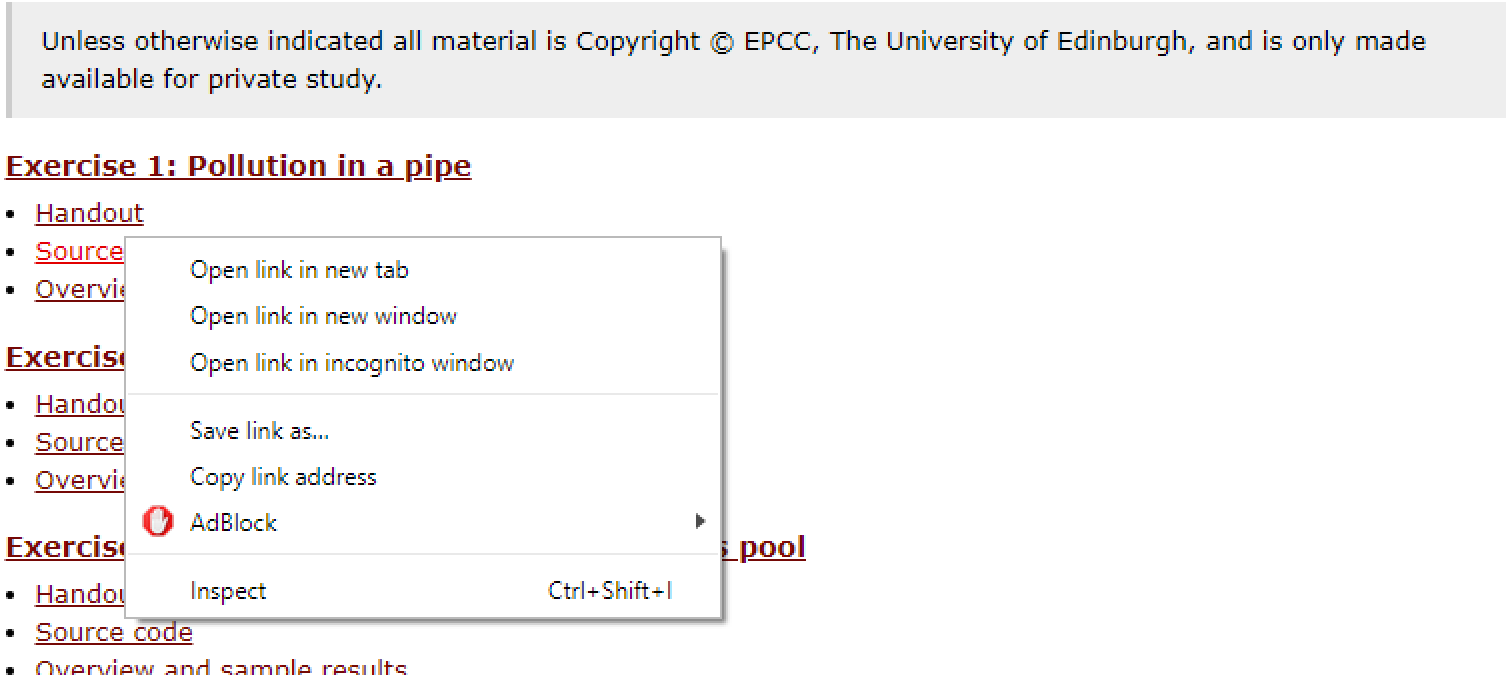
Effectively answering the question how best to combine these individual tasks together to form larger, UE based, groups. In this case we are combining rows of individual tasks with a specific number of rows per UE

Decomposed domain and halos



Getting the source code

http://www.archer.ac.uk/training/course-material/2019/08/parallel-patterns-durham/



Unless otherwise indicated all material is Copyright © EPCC, The University of Edinburgh, and is only made available for private study.

Exercise 1: Pollution in a pipe

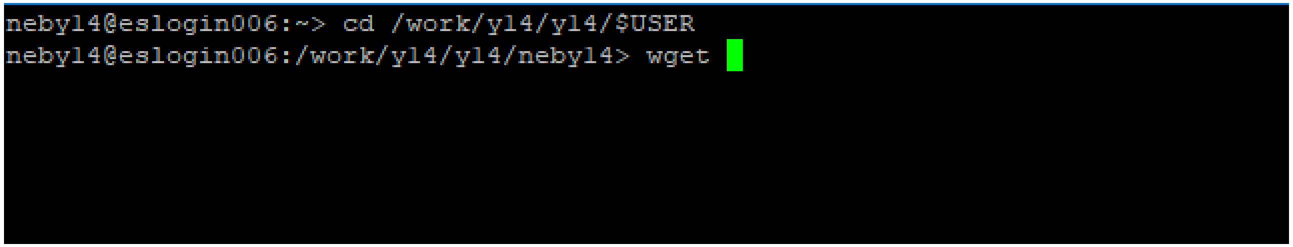
- Handout
- Source
- Overvi

Exercis

- Handou
- Source
- Overvi

Exercis s pool

- Handou
- Source code
- Overview and sample results



```
nebyl4@eslogin006:~> cd /work/y14/y14/$USER
nebyl4@eslogin006:/work/y14/y14/nebyl4> wget
```

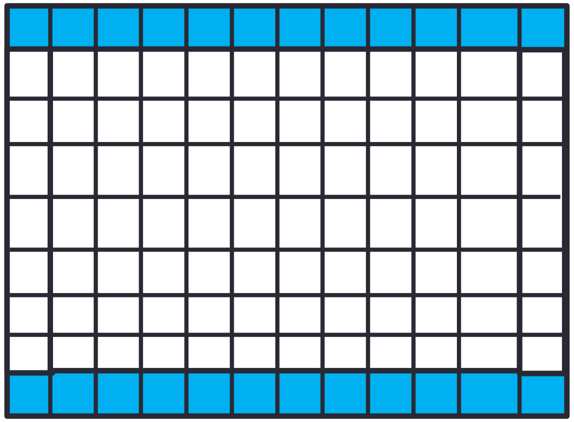
qsub –q [RESERVATION CODE] subpractical.pbs

Wrap up……

Version	Runtime	Speed up	PE
Serial	39.69 s	-	-
Blocking parallel	3.05 s	13.01	0.13
Non-blocking parallel	0.85 s	46.69	0.48
Overlapping parallel	0.81 s	49	0.51

With a domain of size x=1024, y=8192. Solving to 3e-3 relative residual. Parallel runs all done with 4 nodes (96 processes.)

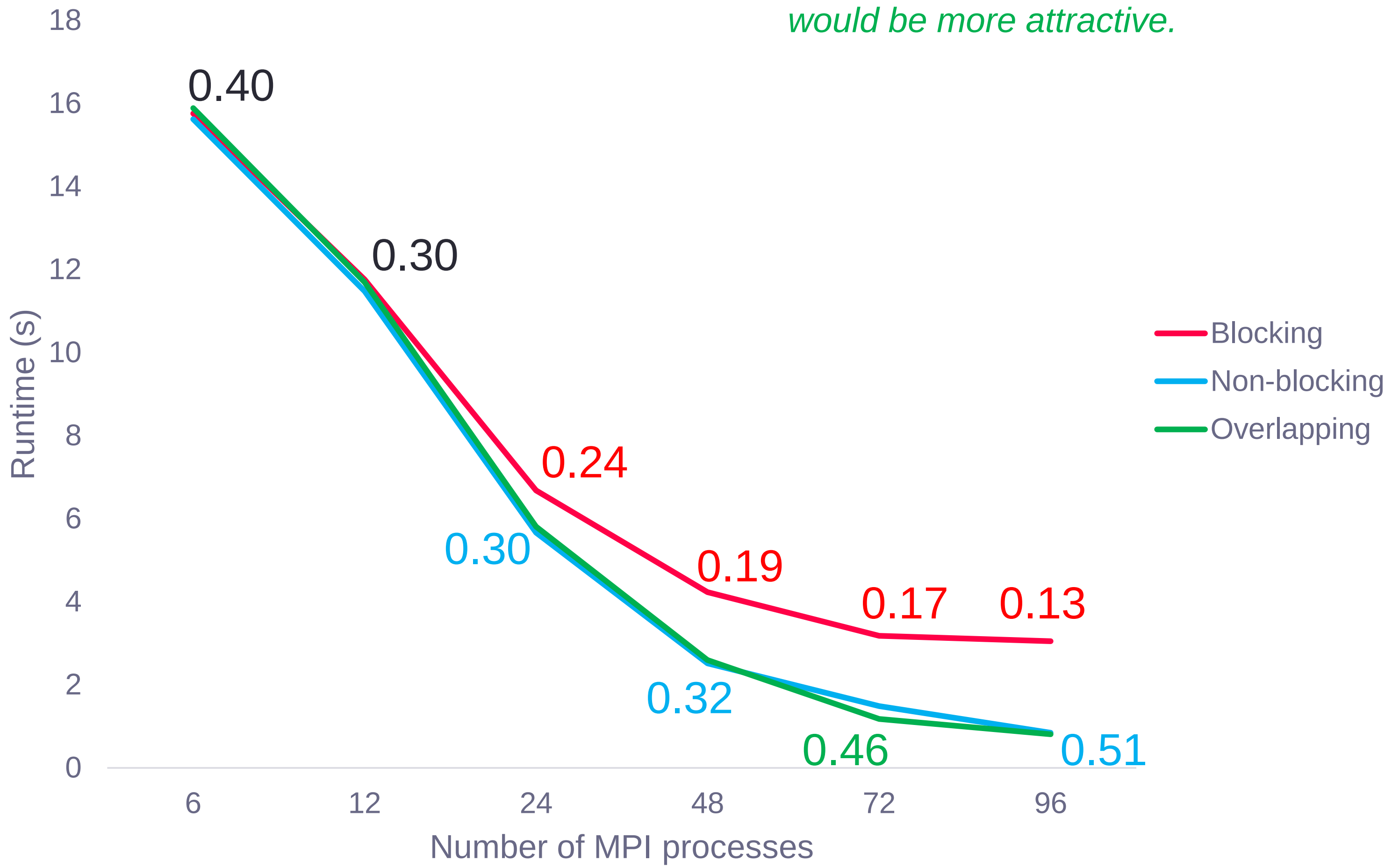
- Going from blocking to non-blocking makes a big difference!
  - Not so much on the overlapping
- Why is this?
  - Not just the behaviour of the different versions, but the PE is quite low!

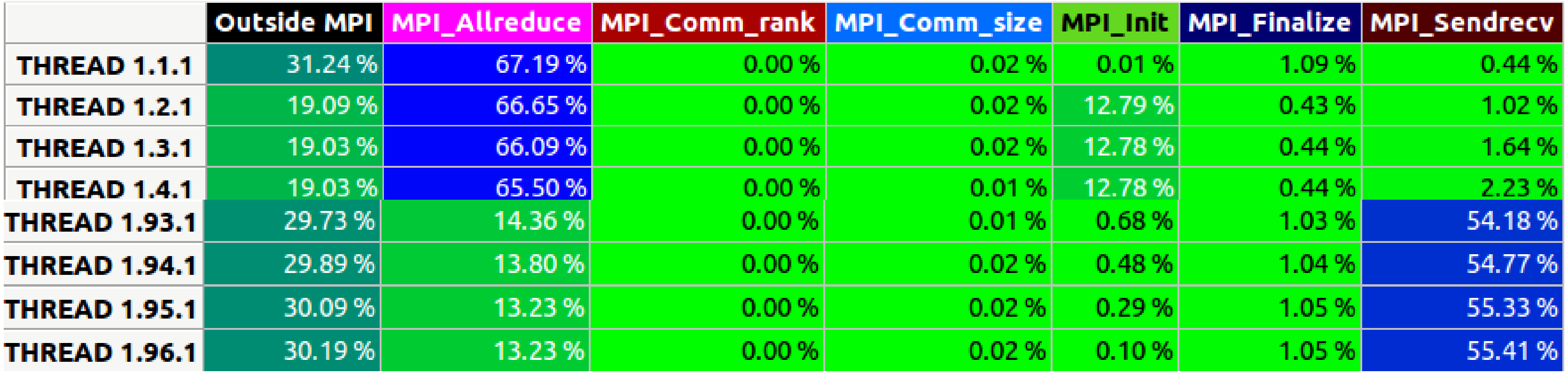


Sample solutions to the code parts are in the solutions directory

Scaling

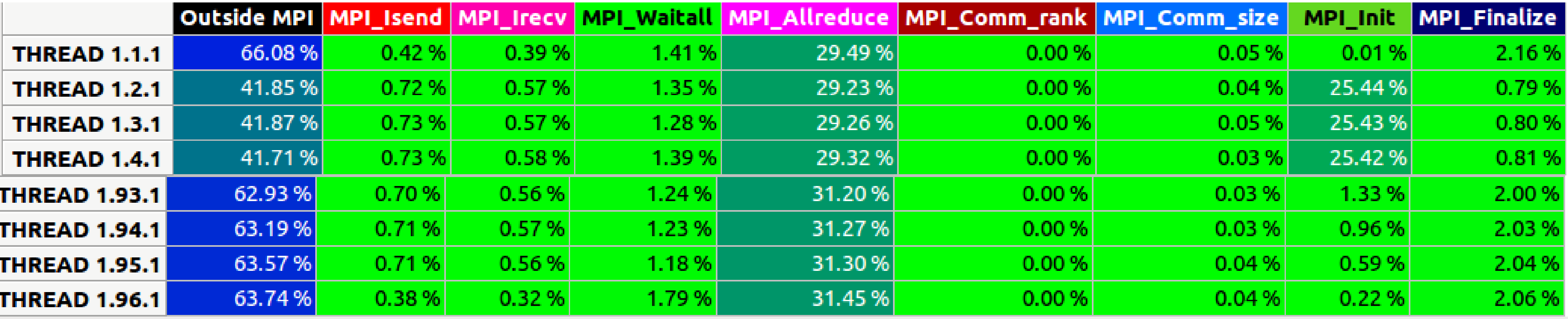
For the parallel efficiency we are comparing against the serial version. The parallel versions running on 1 core take longer (around 55 seconds!) and-so compared to that the PE would be more attractive.





	Outside MPI	MPI_Allreduce	MPI_Comm_rank	MPI_Comm_size	MPI_Init	MPI_Finalize	MPI_Sendrecv
THREAD 1.1.1	31.24 %	67.19 %	0.00 %	0.02 %	0.01 %	1.09 %	0.44 %
THREAD 1.2.1	19.09 %	66.65 %	0.00 %	0.02 %	12.79 %	0.43 %	1.02 %
THREAD 1.3.1	19.03 %	66.09 %	0.00 %	0.02 %	12.78 %	0.44 %	1.64 %
THREAD 1.4.1	19.03 %	65.50 %	0.00 %	0.01 %	12.78 %	0.44 %	2.23 %
THREAD 1.93.1	29.73 %	14.36 %	0.00 %	0.01 %	0.68 %	1.03 %	54.18 %
THREAD 1.94.1	29.89 %	13.80 %	0.00 %	0.02 %	0.48 %	1.04 %	54.77 %
THREAD 1.95.1	30.09 %	13.23 %	0.00 %	0.02 %	0.29 %	1.05 %	55.33 %
THREAD 1.96.1	30.19 %	13.23 %	0.00 %	0.02 %	0.10 %	1.05 %	55.41 %

Blocking parallel version



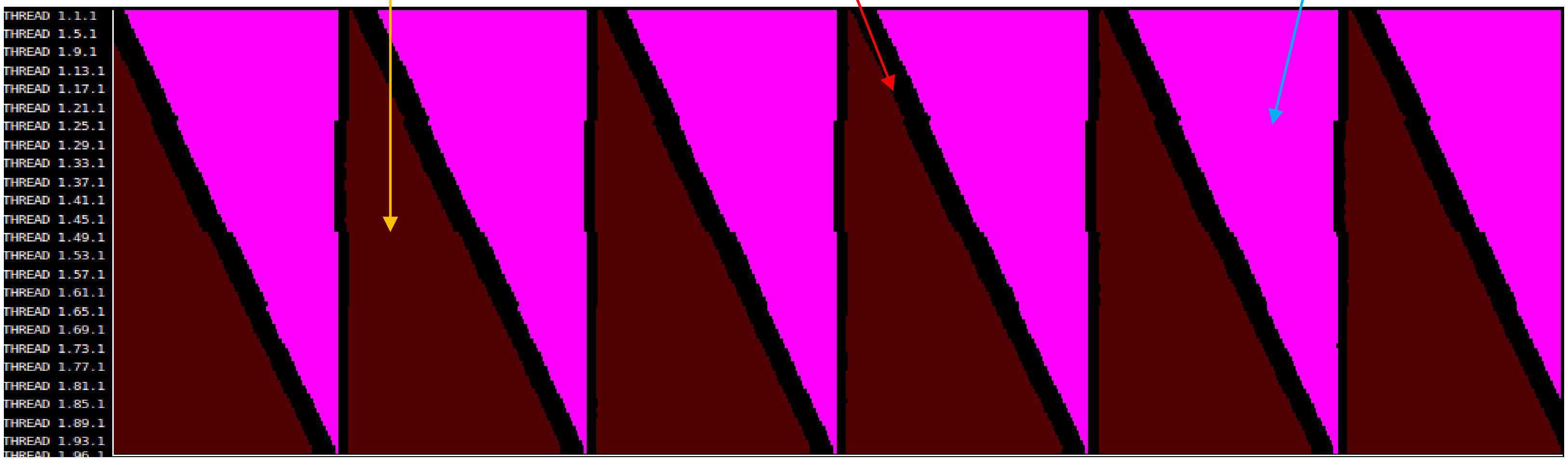
	Outside MPI	MPI_Isend	MPI_Irecv	MPI_Waitall	MPI_Allreduce	MPI_Comm_rank	MPI_Comm_size	MPI_Init	MPI_Finalize
THREAD 1.1.1	66.08 %	0.42 %	0.39 %	1.41 %	29.49 %	0.00 %	0.05 %	0.01 %	2.16 %
THREAD 1.2.1	41.85 %	0.72 %	0.57 %	1.35 %	29.23 %	0.00 %	0.04 %	25.44 %	0.79 %
THREAD 1.3.1	41.87 %	0.73 %	0.57 %	1.28 %	29.26 %	0.00 %	0.05 %	25.43 %	0.80 %
THREAD 1.4.1	41.71 %	0.73 %	0.58 %	1.39 %	29.32 %	0.00 %	0.03 %	25.42 %	0.81 %
THREAD 1.93.1	62.93 %	0.70 %	0.56 %	1.24 %	31.20 %	0.00 %	0.03 %	1.33 %	2.00 %
THREAD 1.94.1	63.19 %	0.71 %	0.57 %	1.23 %	31.27 %	0.00 %	0.03 %	0.96 %	2.03 %
THREAD 1.95.1	63.57 %	0.71 %	0.56 %	1.18 %	31.30 %	0.00 %	0.04 %	0.59 %	2.04 %
THREAD 1.96.1	63.74 %	0.38 %	0.32 %	1.79 %	31.45 %	0.00 %	0.04 %	0.22 %	2.06 %

Non-blocking parallel version

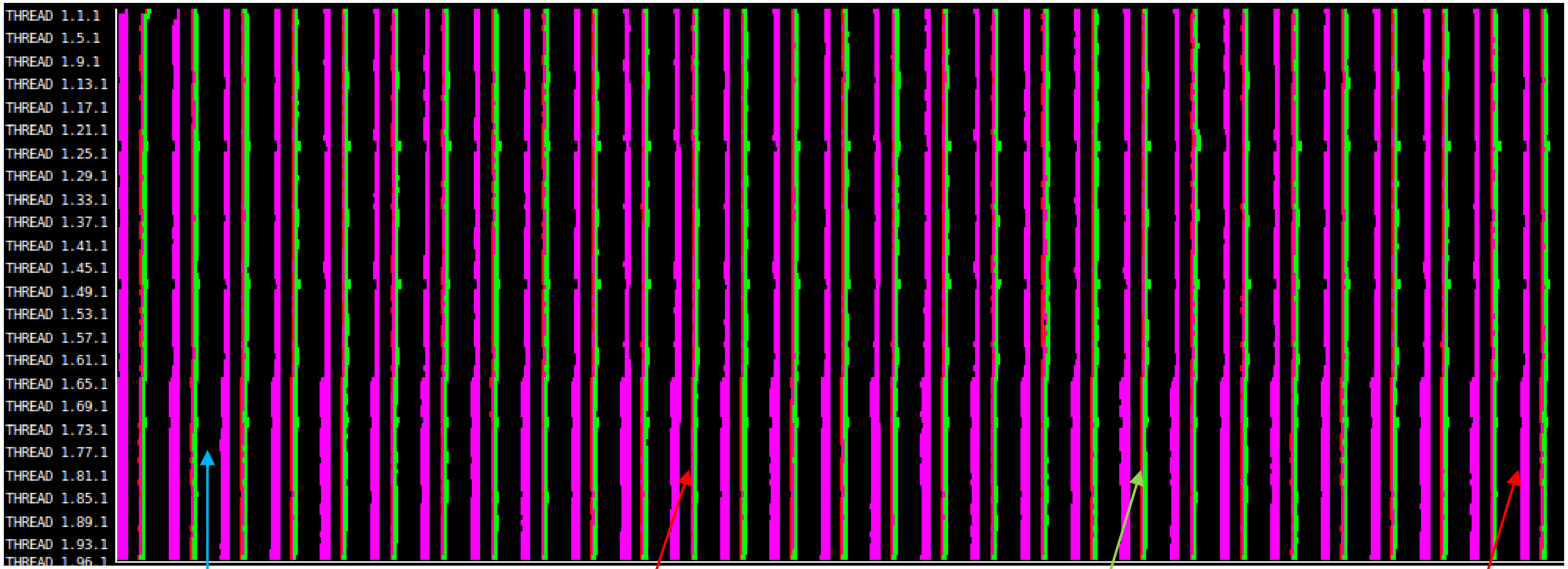
Sendrecv

Computation

Allreduce



Blocking parallel version



Non-blocking parallel version

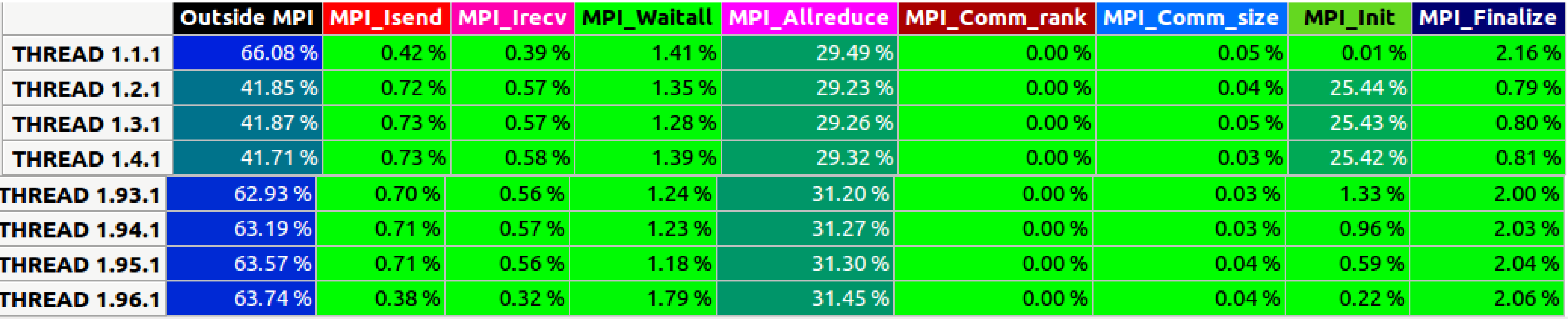
Computation

Isend
Irecv

Waitall

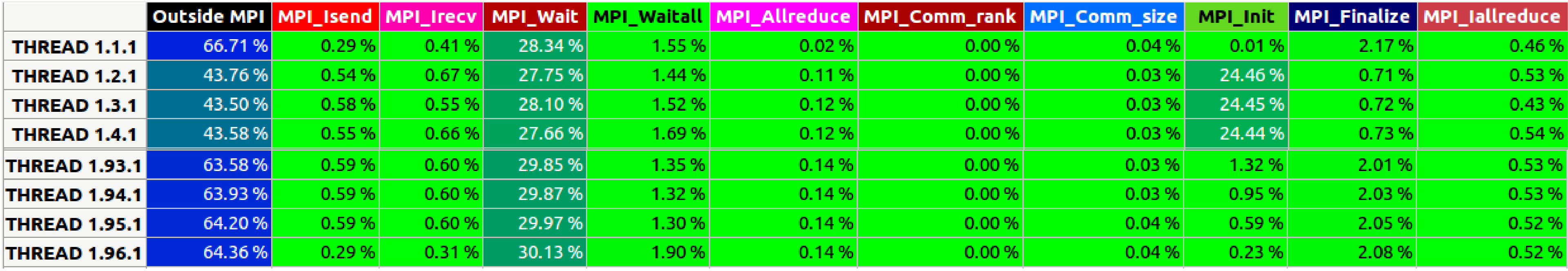
Allreduce

Non-blocking vs overlapping



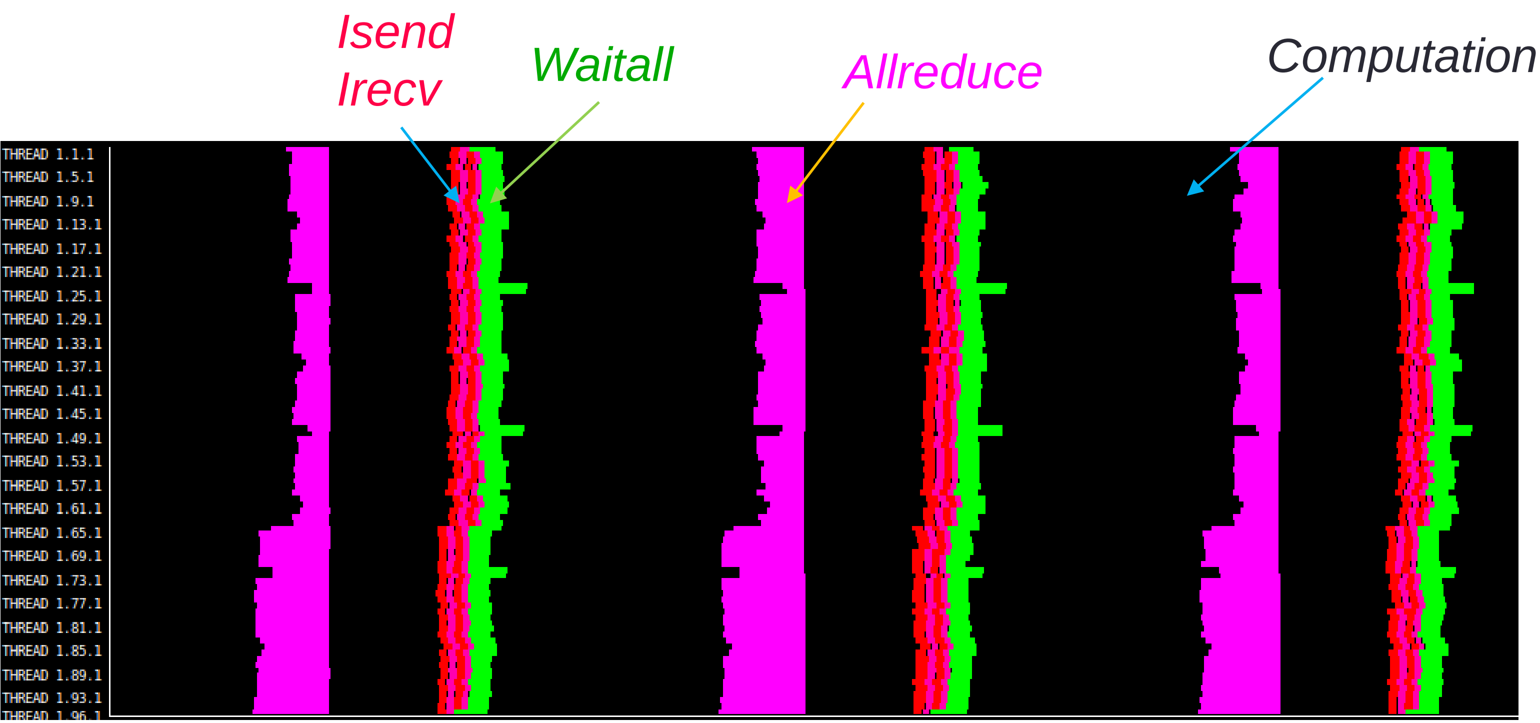
	Outside MPI	MPI_Isend	MPI_Irecv	MPI_Waitall	MPI_Allreduce	MPI_Comm_rank	MPI_Comm_size	MPI_Init	MPI_Finalize
THREAD 1.1.1	66.08 %	0.42 %	0.39 %	1.41 %	29.49 %	0.00 %	0.05 %	0.01 %	2.16 %
THREAD 1.2.1	41.85 %	0.72 %	0.57 %	1.35 %	29.23 %	0.00 %	0.04 %	25.44 %	0.79 %
THREAD 1.3.1	41.87 %	0.73 %	0.57 %	1.28 %	29.26 %	0.00 %	0.05 %	25.43 %	0.80 %
THREAD 1.4.1	41.71 %	0.73 %	0.58 %	1.39 %	29.32 %	0.00 %	0.03 %	25.42 %	0.81 %
THREAD 1.93.1	62.93 %	0.70 %	0.56 %	1.24 %	31.20 %	0.00 %	0.03 %	1.33 %	2.00 %
THREAD 1.94.1	63.19 %	0.71 %	0.57 %	1.23 %	31.27 %	0.00 %	0.03 %	0.96 %	2.03 %
THREAD 1.95.1	63.57 %	0.71 %	0.56 %	1.18 %	31.30 %	0.00 %	0.04 %	0.59 %	2.04 %
THREAD 1.96.1	63.74 %	0.38 %	0.32 %	1.79 %	31.45 %	0.00 %	0.04 %	0.22 %	2.06 %

Non-blocking parallel version

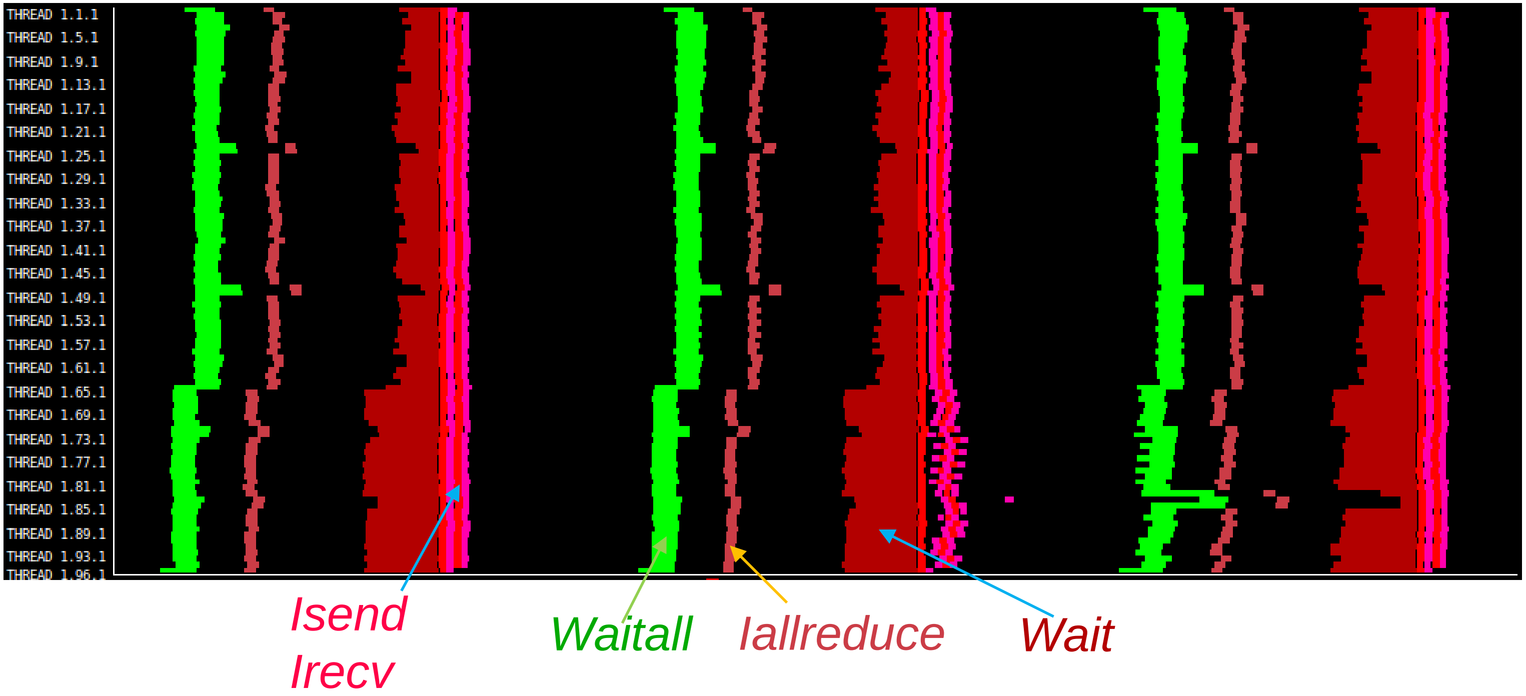


	Outside MPI	MPI_Isend	MPI_Irecv	MPI_Wait	MPI_Waitall	MPI_Allreduce	MPI_Comm_rank	MPI_Comm_size	MPI_Init	MPI_Finalize	MPI_Iallreduce
THREAD 1.1.1	66.71 %	0.29 %	0.41 %	28.34 %	1.55 %	0.02 %	0.00 %	0.04 %	0.01 %	2.17 %	0.46 %
THREAD 1.2.1	43.76 %	0.54 %	0.67 %	27.75 %	1.44 %	0.11 %	0.00 %	0.03 %	24.46 %	0.71 %	0.53 %
THREAD 1.3.1	43.50 %	0.58 %	0.55 %	28.10 %	1.52 %	0.12 %	0.00 %	0.03 %	24.45 %	0.72 %	0.43 %
THREAD 1.4.1	43.58 %	0.55 %	0.66 %	27.66 %	1.69 %	0.12 %	0.00 %	0.03 %	24.44 %	0.73 %	0.54 %
THREAD 1.93.1	63.58 %	0.59 %	0.60 %	29.85 %	1.35 %	0.14 %	0.00 %	0.03 %	1.32 %	2.01 %	0.53 %
THREAD 1.94.1	63.93 %	0.59 %	0.60 %	29.87 %	1.32 %	0.14 %	0.00 %	0.03 %	0.95 %	2.03 %	0.53 %
THREAD 1.95.1	64.20 %	0.59 %	0.60 %	29.97 %	1.30 %	0.14 %	0.00 %	0.04 %	0.59 %	2.05 %	0.52 %
THREAD 1.96.1	64.36 %	0.29 %	0.31 %	30.13 %	1.90 %	0.14 %	0.00 %	0.04 %	0.23 %	2.08 %	0.52 %

Overlapping parallel version

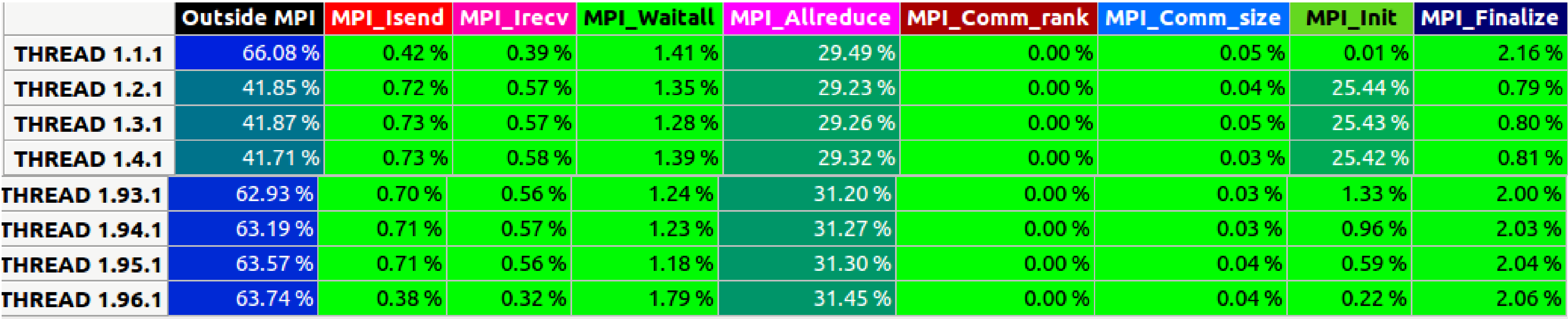


Non-blocking parallel version



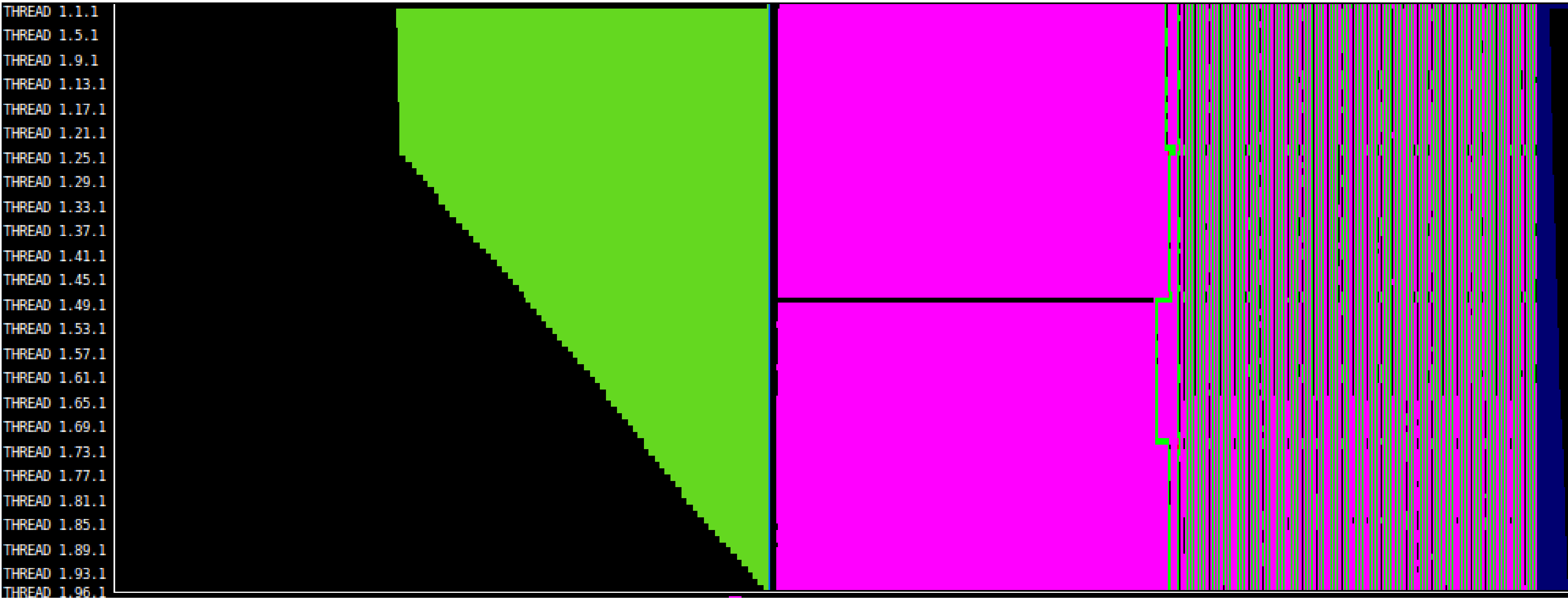
Overlapping parallel version

Why does the utilisation of later ranks seem better with non-blocking version?



	Outside MPI	MPI_Isend	MPI_Irecv	MPI_Waitall	MPI_Allreduce	MPI_Comm_rank	MPI_Comm_size	MPI_Init	MPI_Finalize
THREAD 1.1.1	66.08 %	0.42 %	0.39 %	1.41 %	29.49 %	0.00 %	0.05 %	0.01 %	2.16 %
THREAD 1.2.1	41.85 %	0.72 %	0.57 %	1.35 %	29.23 %	0.00 %	0.04 %	25.44 %	0.79 %
THREAD 1.3.1	41.87 %	0.73 %	0.57 %	1.28 %	29.26 %	0.00 %	0.05 %	25.43 %	0.80 %
THREAD 1.4.1	41.71 %	0.73 %	0.58 %	1.39 %	29.32 %	0.00 %	0.03 %	25.42 %	0.81 %
THREAD 1.93.1	62.93 %	0.70 %	0.56 %	1.24 %	31.20 %	0.00 %	0.03 %	1.33 %	2.00 %
THREAD 1.94.1	63.19 %	0.71 %	0.57 %	1.23 %	31.27 %	0.00 %	0.03 %	0.96 %	2.03 %
THREAD 1.95.1	63.57 %	0.71 %	0.56 %	1.18 %	31.30 %	0.00 %	0.04 %	0.59 %	2.04 %
THREAD 1.96.1	63.74 %	0.38 %	0.32 %	1.79 %	31.45 %	0.00 %	0.04 %	0.22 %	2.06 %

Non-blocking parallel version


THREAD 1.1.1
THREAD 1.5.1
THREAD 1.9.1
THREAD 1.13.1
THREAD 1.17.1
THREAD 1.21.1
THREAD 1.25.1
THREAD 1.29.1
THREAD 1.33.1
THREAD 1.37.1
THREAD 1.41.1
THREAD 1.45.1
THREAD 1.49.1
THREAD 1.53.1
THREAD 1.57.1
THREAD 1.61.1
THREAD 1.65.1
THREAD 1.69.1
THREAD 1.73.1
THREAD 1.77.1
THREAD 1.81.1
THREAD 1.85.1
THREAD 1.89.1
THREAD 1.93.1
THREAD 1.96.1