The Computer as a Solution Space

Problem presented by

Zenotech

Dr David Standingford

ESGI107 was jointly hosted by
The University of Manchester
Smith Institute for Industrial Mathematics and System Engineering

with additional financial support from
KTN Ltd
Natural Environment Research Council
Manchester Institute for Mathematical Sciences

Report author

Harry Braviner (University of Cambridge)
Dr Ostap Hryniv (University of Durham)
Dr Bernard Piette (University of Durham)
Anna Railton (Smith Institute)

Executive Summary

Zenotech provides cloud-based high performance computing (HPC) services to customers wanting to run large parallelised simulations. The transfer of data between nodes in a parallelised problem takes a few orders of magnitude more time than the calculation of the data itself, causing a bottleneck. Any improvements to algorithm implementation that can improve efficiency, reliability, accuracy or robustness to errors would mean more effective and economic use of HPC resource. Zenotech challenged the study group to find a mathematical characterisation of a HPC system that would allow for a more effective mapping of algorithm topology onto HPC network topology and a better understanding of scalability and robustness in the system. We investigated how to best optimise partition size and time of data transfer on a square lattice and the effect of using old boundary data for continued calculation when new data does not arrive on time. We also produced a toy network model to test the effect of different partition choices and communication strategies.

Version 1.0
April 20, 2015
iv+22 pages

Contributors

Harry Braviner (University of Cambridge)
Robert Davey (University of Manchester)
Dr Ostap Hryniv (University of Durham)
Jeremy Minton (University of Cambridge)
Piotr Morawiecki (Polish Academy of Sciences)
Dr Bernard Piette (University of Durham)
Prof. Colin Please (University of Oxford)
Anna Railton (Smith Institute)

Contents

1 Introduction

1.1 Background

(1.1) High performance computing (HPC) is moving towards exascales, with networks containing many thousands of nodes working in parallel. However, little thought is given to the HPC system architecture when solving mathematical problems. Features of HPCs such as latency, bandwidth and memory are seen as limiters that are assumed to be dealt with by a black box compiler and not by algorithm design nor more efficient allocation of work across processors. In general, users of HPC (mathematicians or otherwise) have a limited understanding of the nature and limitations of computing power and this can lead to HPC not being used to its full potential.

(1.2) Zenotech provides cloud-based HPC services, making unlimited HPC resources available to companies without them having to purchase, maintain and manage hardware. End users are also removed from requiring a deep understanding of the working of HPC. Zenotech and the industry as a whole will benefit from any method to maximise the resources available and to increase reliability and accuracy of the computations run on their systems.

(1.3) In large computations, the bottleneck is the time to send data between nodes through the network. The speed of processors (measured in floating-point operations per second, or FLOPs) is more than adequate, summarised in the slogan *'FLOPs are free'*. However, processing speed is constrained by an architecture that needs too much energy to send, receive and store data. It is not currently capable of operating to capacity due to limited local access to memory, as shown in Figure 1.

(1.4) There is scope for more finesse in how algorithms are designed and how work is allocated within a HPC network topology. When parallelising a large computing job, the solution space is split into *partitions*, each of which then becomes the responsibility of a particular node.

(1.5) Currently, processors are 'load balanced', distributing computing workload (split into these partitions) across multiple nodes to optimise resource use. However, this can mean that as nodes finish blocks of processing concurrently, they try to transmit their data simultaneously across the network. This causes delays as the network becomes congested and processors are left hanging while they wait for data from their neighbours.

(1.6) Zenotech would ultimately like to improve the utilisation of computing as a tool for solving mathematical problems. This could be achieved by improving the a-priori **characterisation** of a computer for specific mathematical purposes. There is also a need to improve the **scalability** of very large scale simulations; currently doubling the number of nodes does not make your

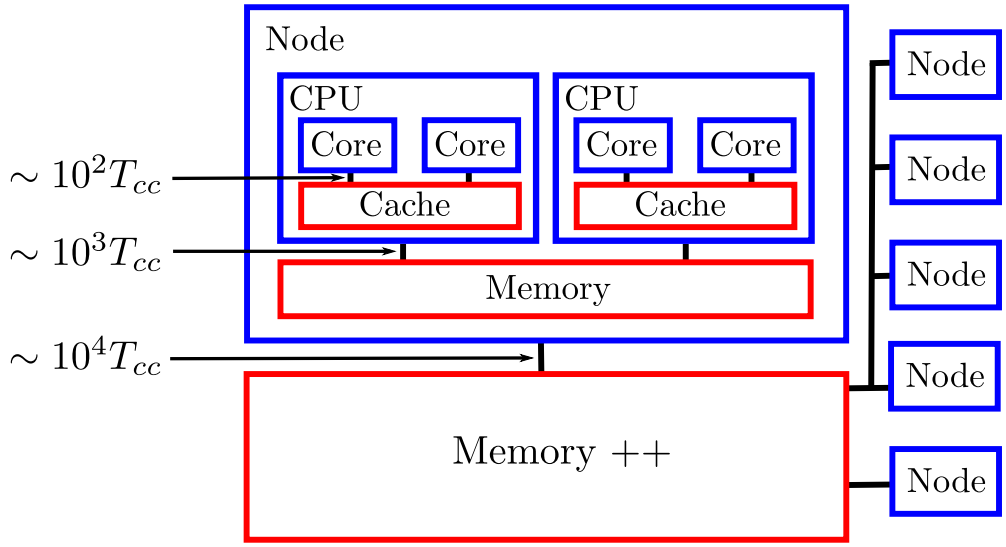

Figure 1: Structure of CPUs inside a node and nodes inside a HPC. There will typically be 8+ cores in a CPU and multiple CPUs in a node. There can be thousands or tens of thousands of nodes in a HPC network. Orders of magnitude for the times to transfer data between different memory locations are in units of clock cycles, T_{cc}, the time needed for a simple CPU operation, and are shown next to the layer of connections they refer to. By far the most costly is the network connection between nodes.

simulation twice as fast. In particular, there is need to improve knowledge of both weak and strong scaling, where:

- **Weak scaling** – how the solution time varies with the number of processors for a fixed problem size *per processor*;
- **Strong scaling** – how the solution time varies with the number of processors for a fixed *total* problem size.

(1.7) Furthermore, as the number of nodes and thus the probability of hardware failure increases, there will be an increased demand for algorithm **robustness** to node failure.

(1.8) There is also the potential for better use of highly distributed computing systems where the mathematics will automatically degrade to normal-form local approximations when data does not arrive in a timely manner.

1.2 Problem statement

(1.9) Is it possible to describe various aspects of computer architecture mathematically and exploit HPC network design to increase system performance, accuracy and robustness?

1.3 Objectives

(1.10) There are many possible directions for tackling this problem, which we discuss in this section. Firstly, can we do **domain decomposition** in a more intelligent way (both from the outset and dynamically as the code is running)? Is it possible to build into the partition size 'breathing space' for potentially late information from other nodes? For example, could you tell `METIS`[1] to map partitions like the topology of a network, putting nodes local to each other in the simulation on nodes connected by low latency?

(1.11) Can we mask communications better by **sending outermost data first**? Data for other nodes does not all have to be calculated and sent in one go; a staggered approach may be useful.

(1.12) Can we learn more **knowledge about the network** as the code proceeds and use this knowledge to our advantage? The network path between any two nodes is not known, cannot be directly discovered and is not always the same between HPCs. However, there is the potential to learn about latencies between nodes and use this information to our advantage in some way.

(1.13) Can we show that some sort of **asynchronous communications protocol** is a more sensible approach than the even loading of CPUs? What is the best way of scheduling communications so they don't overlap and how can we do this robustly, given the unknowns in the network? Is it possible to achieve a uniform rate of requests/transfers across the network?

(1.14) Can we create a **toy network model** and input known partition metadata to test new ways of partitioning the system? This model can be validated against known HPC behaviour and timings. There could also be the possibility of including both asynchronous behaviour and network uncertainty.

(1.15) **'Egalitarian model of data transfer'** – is it possible to make intelligent decisions on whether to send data from node P_i through the network depending on whether other nodes are actually waiting for P_i's data? This is a potential paradigm shift as instead of thinking about what node P_i needs, we could think about what P_i can provide to other nodes. Could this benevolence model of asking for data instead of just exchanging when ready, improve performance?

(1.16) **Dependence on PDE type** – can we represent hyperbolic, elliptic or parabolic systems in a 'normal form' based upon computer architecture? Can we apply a similar analysis to spectral methods?

(1.17) **Robustness** – what is the best way to dissipate out errors? Can an algorithm 'struggle on' in the face of the loss of any one node? Is it possible to identify the most critical nodes at any one point and build in some redundancy? There will be some dependency on the problem type (hyperbolic,

[1] METIS is graph partitioning software which is widely used in HPC.

elliptic etc.) as to how catastrophic node failure is. This could be investigated using a toy network model or by numerical analysis of different PDE types and algorithms.

(1.18) It was decided to consider graphics processing units (GPUs) as just a system with many CPUs and different parameters and thus not as separate case.

1.4 Overview of our approach

(1.19) Over the week, our approach to these problems was based around three main directions of enquiry. In Section 2.1 we attempt to quantify when it is most efficient to transfer data between nodes. We aim to optimise the number of partitions and the timesteps between copying boundary data to other nodes. We do this in different dimensions d and for differing network latencies and bandwidths.

(1.20) We propose a new way of thinking about how to parametrise algorithm efficiency in Section 2.2, where we consider computing nodes as vertices in a directed graph.

(1.21) In Section 2.3, we consider the impact of data arriving late to a node. Can a node realistically keep calculating with old data while waiting?

(1.22) We also produced a toy model of a network made of switches, nodes and processors, which could be used as a test bed for the effectiveness of different algorithm implementations without the need for running jobs on HPCs. The details of this can be found in Section 2.4.

2 Approaches to the problem

(2.23) We are considering an unsteady problem, where a solution on a d–dimensional spatial grid is propagated forward in time by an explicit method, i.e. a classical finite difference time domain. A similar analysis could be done for a steady problem, a method where harmonic time–dependence had been removed, or a method using implicit time–stepping.

2.1 Optimising strategy on a square lattice

(2.1) In this section we assume we are using an integration scheme that is local and only requires a relatively small number of data points to be transferred between partitions after each integration step.

(2.2) We will split the domain into N_p square partitions, each of size L^d, where d is the spatial dimension. We give definitions used in this section in Table 1, but

M	size of domain (total number of grid points)
d	spatial dimension
N_p	number of partitions, $N_p^{(opt)}$ is the optimal value
$L = (M/N_p)^{1/d}$	side length of each partition hypercube
T_{CPU}	time needed to compute one integration time step on one grid point
w	number of neighbours required on each side of a node for one time step of the integration, $w \geq 1$. This depends on the integration method.
T_{lat}	the latency of the network before each network transaction
D_L	area of partition that does not depend on data from adjacent partitions
D_B	area of partition that needs to be integrated before boundary data can be copied to the neighbours
B	available network bandwidth
B_0	bandwidth available to each node when the network is not used
B_{sat}	total maximum bandwidth across the whole network
k	timesteps between copying boundary data between nodes, $k^{(opt)}$ is the optimal value

Table 1: List of parameters used in Section 2.1

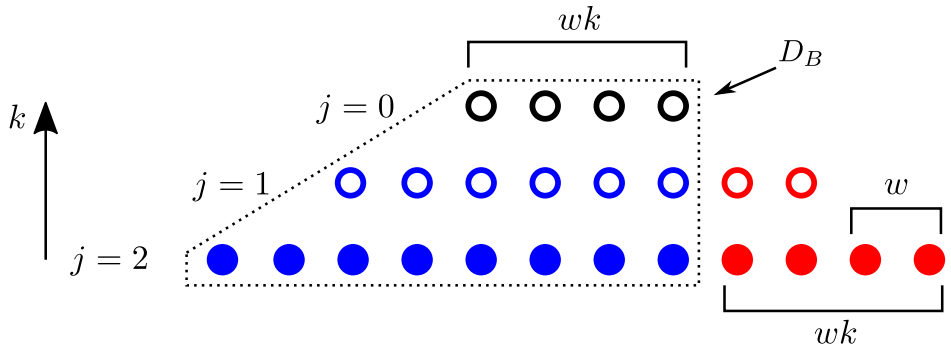

Figure 2: Integration at the edges ($k = 2, w = 2$). The blue points are the responsibility of one node and the red values are those that need to be transmitted from the adjacent partition before the blue node does its k timesteps. Timestepping is from $j = k$ to $j = 0$. The black circles are the values that will need transmitting from the blue node to the red after these k timesteps before the next step.

we often refer to the partitions as squares or cubes, rather than hypercubes.

(2.3) After k integration steps, we will need to copy some data at the boundary of each partition between nodes so that each node can carry out a few integration steps without further data. We can view the local partition as a set of nested concentric cubic layers and what we have to consider is the number of layers, $w = w_n$, that must be copied between partitions. We are assuming that the integration scheme evaluates the solution at time $t + 1$ in terms of values at time t.

(2.4) Each partition can be split in two regions:

- D_L is the area at the centre which does not depend on the data coming from the adjacent partitions;
- D_B is the layers near the boundary which will need to be integrated before the boundary data can be copied to the neighbours.

(2.5) In the simplest case, $w = k = 1$, we only need to copy one layer of data that is all the data at the boundary of the cube. D_B will then be two layers deep and D_L will be the entire domain minus the two outermost layers. If $k > 1$, then D_B has a trapezoidal shape as depicted in Figure 2.

(2.6) The operations that need to be performed are the following:

1. Solve the equation on the edges of the domain for k steps before data can be copied between nodes.
2. While transfer takes place, the equation will be solved inside the domain and we assume that this takes less time than the network transfer.
3. If not, the bottleneck is the computation and the partition number N_p can be increased.

(2.7) The time needed to perform k steps is thus the sum of the time needed to solve the problem on the edges (blue and red points in Figure 2) and the time needed to copy the data between the adjacent partitions (filled red points in Figure 2).

(2.8) If we look at each step, the number of computations that must be performed near the edges is given by

$$n_j = (L + 2wj)^d - (L - 2kw - 2wj)^d \approx 2dL^{d-1}(2wj + kw) \tag{1}$$

and if we sum over the k steps,

$$\sum_{j=0}^{k-1} n_j = 2dL^{d-1}w(2k^2 + k). \tag{2}$$

(2.9) So, the time needed to perform the k integration steps is

$$2dL^{d-1}w(2k^2 + k)T_{CPU}, \tag{3}$$

where T_{CPU} is the time needed to compute one integration time step on one grid point.

(2.10) The volume of data that must be copied after k steps will be

$$2dL^{d-1}wk. \tag{4}$$

This needs to be done *twice*: once to copy local data to each neighbour and once to read data outside the domain from the neighbours.

(2.11) Therefore, the time required to copy the data is

$$4dT_{lat} + 4dL^{d-1}wk/B, \tag{5}$$

where B is the available network bandwidth and T_{lat} is the latency of the network before each network transaction. It has a factor of $4d$ as each partition has $2d$ neighbours and it needs to both read and copy local data from and to each of these. If this is only done once every k timesteps, this will be divided by k, as in equation (8).

(2.12) When the network is not used, each node will have a bandwidth B_0 but as the network is saturated, they will only get a fraction of that.

(2.13) If B_{sat} is the total maximum bandwidth across the whole network, the bandwidth available, on average, is assumed to be B_{sat}/N_p.

(2.14) So putting all this together, we get

$$t_{k\,step} = \max\left[\left(k\frac{M}{N_p} + 2dL^{d-1}wk(k+1)\right)T_{CPU}, \quad 2dL^{d-1}w(2k^2+k)T_{CPU} + 4dT_{lat} + \frac{4dL^{d-1}wkN_p}{B_{sat}}\right] \tag{6}$$

$$= \max\left[\left(k\frac{M}{N_p} + 2d\frac{M^{(d-1)/d}}{N_p^{(d-1)/d}}wk(k+1)\right)T_{CPU}, \quad 2d\frac{M^{(d-1)/d}}{N_p^{(d-1)/d}}w(2k^2+k)T_{CPU} + 4dT_{lat} + \frac{4dM^{(d-1)/d}wN_p^{1/d}k}{B_{sat}}\right], \tag{7}$$

using the definition of L given in Table 1.

(2.15) The first argument of the max function is the total time needed to compute all the lattice points on the partition, including the red points in Figure 2. The second argument is the time needed to compute all the lattice points on the edges of the lattice and to copy them to the $2d$ neighbours as well as to obtain copies of data from the neighbours.

(2.16) Therefore, the time needed to perform one step is

$$t_{1\,step} = \frac{t_{k\,step}}{k} = \max\left[\left(\frac{M}{N_p} + 2d\frac{M^{(d-1)/d}}{N_P^{(d-1)/d}}w(k+1)\right)T_{CPU}, \quad 2d\frac{M^{(d-1)/d}}{N_p^{(d-1)/d}}w(2k+1)T_{CPU} + \frac{4dT_{lat}}{k} + \frac{4dM^{(d-1)/d}wN_p^{1/d}}{B_{sat}}\right], \tag{8}$$

(2.17) We now want to determine the optimum value of N_p, w and k. Before we do this, we will assume that the bulk of the partition is much bigger than the edges or, more explicitly,

$$\frac{M}{N_p} \gg 2d\frac{M^{(d-1)/d}}{N_P^{(d-1)/d}}w(k+1). \tag{9}$$

(2.18) Before we proceed to find how to minimize the time needed to compute each step, we must try evaluate what is likely to be the biggest bottleneck.

(2.19) If we consider a three dimensional problem where each domain is of size $L = 100$, there are 10^6 points inside the domain and if we consider $k = w = 1$ each edge contains about 6×10^4 points. If the network has a bandwidth of 10Gbit, it can transfer about 1GB per second, (gigabyte, 1 byte is 8 bit), and so about 10^8 doubles per second (1 double is 8 bytes). It will thus take

of the order of 10^{-4} to 10^{-3} seconds to copy the data points on the edges. We believe that the latency of the switches can vary between 1ms and 1μs and so the transfer times are right within that interval. In our example we have assumed that we have only one function for our equation when one can easily have 10 or more. This would increase the copying time to 1ms. Using other values of k and w would increase the copying time too.

(2.20) We must also consider how switches work. In our estimation of the network performance in equation (8), we assume that the switches establish the links between nodes once for each transfer. T_{lat} is thus paid only once per data transfer per partition side. Is this really how a switch works or are the data split into blocks for which the connection must be re-established each time? If the latter applies, then the switch latency will be applied to every block of data and becomes proportional to the amount of data transferred.

(2.21) In equation (8), B_{sat} is the total bandwidth available on the network and we have assumed for the sake of simplicity that it is, on average, shared equally between all the nodes. If the switch latency is applied to each data block as described above, thus inducing a time penalty proportional to the amount of data transferred, it must be included in the estimation of B_{sat}.

(2.22) In any case, our estimation above suggests that the time needed to copy data between the different partitions will dominate the time needed to perform the calculations.

(2.23) We will now proceed with the estimation of the best parameters for N_p and k to minimize equation (8), $N_p^{(opt)}$ and $k^{(opt)}$.

2.1.1 Limit when the network bandwidth is the main factor

(2.24) If the time needed to compute the equation, $\frac{M}{N_p}T_{CPU}$, is smaller than the transfer time, then

$$2d\frac{M^{(d-1)/d}}{N_p^{(d-1)/d}}w(k+1) \tag{10}$$

is very small too. Moreover, if the latency is smaller than the transfer time, then the optimal value for N_p will be obtained when

$$\frac{MT_{CPU}}{N_p^{(opt)}} = \frac{4dM^{(d-1)/d}wN_p^{1/d}}{B_{sat}} \tag{11}$$

and so

$$N_p^{(opt)} = \left(\frac{MT_{CPU}B_{sat}}{4dM^{(d-1)/d}w}\right)^{d/(d+1)}. \tag{12}$$

2.1.2 Limit when network latency is the main factor

(2.25) In this section we consider the other limit, i.e. where the network latency dominates. As argued above, this will apply if the latency is paid only once per full data transfer and when the edges of the partitions are relatively small (as in two dimensional problems, for example).

(2.26) In this limit, $\frac{M}{N_p}T_{CPU}$ is considered large and the optimal condition is met when

$$\begin{aligned}\frac{M}{N_p}T_{CPU} &= 2dM^{(d-1)/d}N_p^{(1-d)/d}w(2k+1)T_{CPU} \\ &\quad +\frac{4d}{k}T_{lat}+\frac{4dM^{(d-1)/d}wN_p^{1/d}}{B_{sat}}\end{aligned} \tag{13}$$

or

$$\begin{aligned}MT_{CPU} &= 2dM^{(d-1)/d}N_p^{1/d}w(2k+1)T_{CPU} \\ &\quad +\frac{4d}{k}T_{lat}N_p+\frac{4dM^{(d-1)/d}wN_p^{(1+d)/d}}{B_{sat}},\end{aligned} \tag{14}$$

which can be solved for specific dimension d.

(2.27) However, if B_{sat} is very large, and the time needed to compute all the lattice points inside the partition domain is larger than the one needed for the edges, we can neglect the first and third term on the right hand side of equation (15) to obtain

$$\frac{M}{N_p}T_{CPU}=\frac{4d}{k}T_{lat} \tag{15}$$

then the optimal number of partitions is

$$N_p^{(opt)}\approx\frac{kMT_{CPU}}{4dT_{lat}}. \tag{16}$$

(2.28) We therefore see that, in this limit, the optimal number of partitions is proportional to the ratio between the total time needed to compute the equation on the full domain of the problem and the switch latency.

(2.29) To optimise k, we impose

$$\frac{dt_{1step}}{dk}=dM^{(d-1)/d}N_p^{(1-d)/d}2wT_{CPU}-\frac{4d}{k^2}T_{lat}=0 \tag{17}$$

and so

$$\begin{aligned}k^{(opt)} &= \left(\frac{4dT_{lat}}{2wT_{CPU}dM^{(d-1)/d}N_p^{(1-d)/d}}\right)^{1/2} \\ &= \left(\frac{2dT_{lat}}{wT_{CPU}dL^{d-1}}\right)^{1/2}.\end{aligned} \tag{18}$$

(2.30) This shows that $k^{(opt)}$ is the square root of the ratio between the network latency time and the time needed to compute on the edges of a partition. One must also remember that k must be an integer and it would thus be inappropriate to substitute (18) into (16).

(2.31) The evaluations above assumed that the system is homogeneous. In practice, the latency will vary between nodes. We can use equation (16) using for T_{lat} the average latency on the network, but then we must balance out the nodes according to their individual latency.

(2.32) We denote $T_{lat,i}$ the average latency of partition i between all the neighbours of that partition, and $M_i = L_i^d$ the partition domain size. The time needed for each step will be governed by the largest latency, T_{lat,i_M}, on the next work on node i_M. We could then increase the volume of any other partition i so that

$$L_i^d T_{CPU} = \frac{4d}{k} T_{lat,i_M} \tag{19}$$

without any loss of speed. The volume of the domain i_M could then be reduced but the gain will be minimum as it will only decrease the time needed to compute the border of the partition which is small compared to the time needed to compute the bulk of the domain. On the other hand, if the partitions are irregular by design, smaller domain should be mapped to the one with the largest latency, but the benefit will be small.

2.1.3 Realistic limit

(2.33) In practice, the latency and the network bandwidth are likely to play a role and one must therefore optimise equation (8). This will lead to algebraic equations which can't be solved analytically but can be evaluated numerically if one has estimates of the different parameters. The first thing to do will be to evaluate if one is close to any of the limits considered in our analysis and then to refine the optimal values of N_p and k numerically.

2.2 Algorithm efficiency for parallelisation

(2.34) An algorithm's efficiency is typically described by how the number of computations scale with the size of the problem. This becomes an insufficient measure once the problem exceeds the capability of a single machine as then data transfer must be considered as well.

(2.35) If we consider the computational process as a **directed graph**, where each node is a computation that requires data from all the computations connected upstream, then we begin to visualise connectedness. More specifically (by applying cuts to this graph), partitions indicate where in computer space

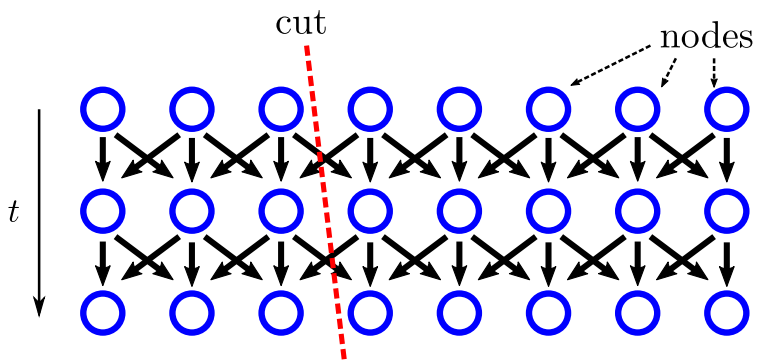

Figure 3: Visualising an algorithm as a graph. Blue circles represent nodes doing some calculation, while arrows between them show the dependence on other nodes before further computation can continue, i.e. information transfers. Time t increases from top to bottom, which each horizontal row of nodes computing the same timestep. Cuts create partitions and thus here we have two partitions.

the computations are performed. The edges passing over a given cut indicate data that must pass between nodes. This visualisation is shown in Figure 3.

(2.36) This then becomes a scheduling/commodity problem to minimise the maximum of the data transfer times and computation times,

$$\texttt{minimax}\Big\{\text{number of vertices in partition} \times \text{computation time},\\ \text{total information transfers} \times \text{transfer time}\Big\}. \tag{20}$$

However, more generally, it is likely that a given algorithm has a natural choice of cuts as well as a high degree of symmetry making many cuts equivalent.

(2.37) This could be exploited with more sophisticated scaling laws that indicate how information transfer scales in addition to computations. For example, letting n denote the size of the problem and M denote the number of computational nodes, a simple central differencing scheme would scale $O(n/M, M)$ where the first term is the computations and the second is data transfer. Compare this to a spectral method which requires a Fourier transform at each step so scales like $O(n \log n, nM)$. Both of these methods would also scale linearly with the number of timesteps.

(2.38) This is a clear indication of the parallelisability of an algorithm and would be a more modern indication of algorithm efficiency.

(2.39) One shortcoming of this method is that it is not obvious how to represent interdependence of computation and data transfer. It is possible to envisage two algorithms with the same computation and data transfer requirements,

but where one is highly flexible in when the data is transferred whereas the other is highly restricted.

(2.40) One idea to incorporate this is to count the number of upstream computations required by each node in the graph and take the difference between the current node's value and the value of the node being transferred from another partition. This would indicate the time difference between the data being available for transfer and the time the data is required at its destination and would therefore provide some measure of an algorithm's time flexibility. Unfortunately, this begins to degrade the elegance and intuition of the proposition.

2.3 Impact of transferred data arriving late

(2.41) During the week, we also considered the impact of a node continuing to calculate with old boundary data when fresh data arrives late.

(2.42) Consider a one-dimensional problem of the type

$$\mathbf{u}_t + a\mathbf{u}_x = d\mathbf{u}_{xx} \tag{21}$$

computed on the grid $\mathcal{N} = \{0, 1, \ldots, \mathsf{n}, \mathsf{n}+1\}$ using the scheme

$$\mathsf{M}^{n+1} = \mathsf{A}\,\mathsf{M}^n\,, \tag{22}$$

where A is the matrix of computation and $\mathsf{M}^n = \left(M_0^n, M_1^n, \ldots, M_{\mathsf{n}}^n, M_{\mathsf{n}+1}^n\right)^{\mathsf{tr}}$ is the discretisation of the solution (M_k^n is the value of the nth iteration at node k).

(2.43) For example, for the heat equation ($a = 0$ and $d = 1$) we would have $\mathsf{A} = (a_{ij})$ with the only non-vanishing entries in row i, $1 \le i \le \mathsf{n}$, being

$$a_{i,i-1} = \rho\,, \quad a_{ii} = 1 - 2\rho\,, \quad a_{i,i+1} = \rho\,, \qquad \rho = \tfrac{\Delta t}{\Delta x^2}\,, \tag{23}$$

while the 0^{th} and the $\mathsf{n}+1^{\mathsf{st}}$ rows are chosen according to the boundary conditions imposed on (21).

(2.44) The pure advection equation ($a = 1$ and $d = 0$) would have A as the first order finite difference (central or asymmetric) chosen, while for general a and d A will be a linear combination of these two cases.

(2.45) If n is so large that the whole vector M of solutions cannot fit on a single computer, one has to split the grid $\mathcal{N}$ into parts and solve separate sub-problems on different nodes while communicating the corresponding 'boundary values' between the nodes. For simplicity we consider an idealized network of just two processors, 'left' and 'right', the generalisation to more complex situations being straightforward.

(2.46) We split the computational grid $\mathcal{N}$ between the left and right nodes as follows

$$\mathcal{L} = \{0, 1, 2, \ldots, \mathsf{i}, \mathsf{i}+1\}\,, \qquad \mathcal{R} = \{\mathsf{i}, \mathsf{i}+1, \ldots, \mathsf{n}, \mathsf{n}+1\}\,,$$

ie., for some i, $1 < \mathsf{i} < \mathsf{n} - 1$, the lattice vertices $\{\mathsf{i}, \mathsf{i}+1\}$ form the common part (or 'boundary') of the two subdomains and thus must belong to both left and right partition sets.

(2.47) Notice that if the values of our solution at iteration n are known at all nodes in $\mathcal{L}$, in the next step only the values in $\mathcal{L} \setminus \{\mathsf{i}+1\}$ (where $B \setminus A = \{x \in B \,|\, x \notin A\}$) can be computed locally, while the value $L^{n+1}_{\mathsf{i}+1}$ must be copied from the corresponding value in $\mathcal{R}$. Similarly, if all values of the solution are known for each lattice vertex in $\mathcal{R}$, in the next step the values of the solution in $\mathcal{R} \setminus \{\mathsf{i}\}$ can be computed locally while the value R^{n+1}_{i} must be copied from the value computed in $\mathcal{L}$.

(2.48) If the copied data arrives late, our scheme will read

$$\begin{aligned} L^{n+1}_j &= L^n_j + f(L^n_{j-1}, L^n_j, L^n_{j+1})\,, \quad j \le \mathsf{i}\,, \qquad L^{n+1}_{\mathsf{i}+1} = R^n_{\mathsf{i}+1}\,, \\ R^{n+1}_k &= L^n_k + f(R^n_{k-1}, R^n_k, R^n_{k+1})\,, \quad k \ge \mathsf{i}+1\,, \qquad R^{n+1}_{\mathsf{i}} = L^n_{\mathsf{i}}\,, \end{aligned} \tag{24}$$

where f is the function used to discretise the spatial part of the problem in equation (21) (like the ρ–dependent part of equation (23) for the heat equation).

(2.49) In the matrix form, this is

$$\begin{bmatrix} \mathsf{L} \\ \mathsf{R} \end{bmatrix}^{n+1} = \widetilde{\mathsf{A}} \begin{bmatrix} \mathsf{L} \\ \mathsf{R} \end{bmatrix}^{n} \tag{25}$$

where the square matrix $\widetilde{\mathsf{A}} = (\tilde{a}_{jk})$ (whose dimension is 2 larger compared to that of A, due to duplicating the grid node $\mathsf{i}+1$ in $\mathcal{L}$ and i in $\mathcal{R}$) is defined via

$$\begin{aligned} \tilde{a}_{j,k} = a_{j,k}\,, \quad 0 \le j \le \mathsf{i}\,, \quad 0 \le k \le \mathsf{n}+1\,, \\ \tilde{a}_{\mathsf{i}+1,\mathsf{i}+3} = \tilde{a}_{\mathsf{i}+2,\mathsf{i}} = 1\,, \\ \tilde{a}_{j+2,k+2} = a_{j,k}\,, \quad \mathsf{i} < j \le \mathsf{n}+1\,, \quad 0 \le k \le \mathsf{n}+1\,, \end{aligned} \tag{26}$$

where all other entries of $\widetilde{\mathsf{A}}$ are set to zero.

(2.50) Consider this problem to be

$$\mathbf{u}^{n+1} = \widetilde{\mathsf{A}}\mathbf{u}^n. \tag{27}$$

We discussed computational linear stability of this scheme (which requires that all eigenvalues of the matrix $\widetilde{\mathsf{A}}$ are smaller than 1 in absolute value). It was numerically implemented in MATLAB and considered for various standard methods and values of a and d in (21). The diffusion operator was considered using central differences and the advection operator was considered

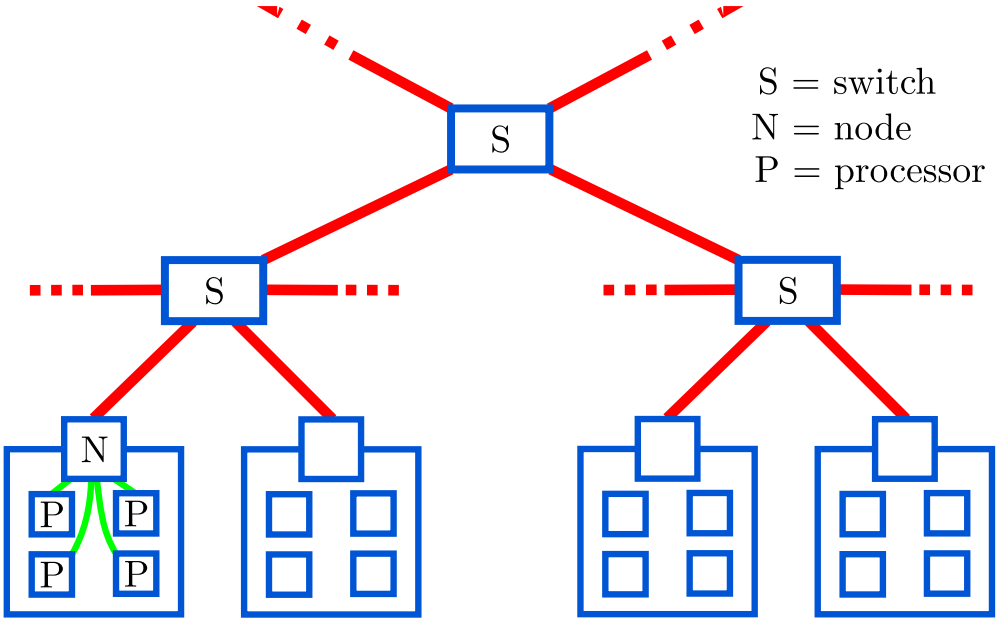

Figure 4: Structure of a HPC network, showing switches connecting nodes. Fast connections (i.e. between processors) are shown in green while slower ones are in red.

using both central differences and upwinding. The preliminary numerical investigations in a number of cases we considered reproduced the conventional results for the case which did not involve information transfer between nodes.

(2.51) For the pure diffusive case the maximum timestep appeared insensitive to introduction of the two-node exchange mechanism; a similar result was unexpectedly found for the advection equation. For intermediate results there appeared situations where slightly larger timesteps might be possible. Hence, these preliminary results indicate that the *latency effects do not reduce the stability regime*.

(2.52) Using the scheme outlined above, the effect on accuracy will be of order of Δt, since at every time the solution is dependent on the information which for some vertices is one step old. We haven't analysed this in detail, but concluded that latency effects will significantly influence those problems where time accuracy is essential.

2.4 `HPCsim` - a Python simulation of a High Performance Computer

(2.53) The source code for the simulation described in this section can be found at:

https://github.com/harrybraviner/HPCsim

2.4.1 Motivation and aims

(2.54) In high performance computation the majority of the wall-clock time is spent on data transfer, rather than computation. This consists of:

- CPU register ↔ cache transfers
- cache ↔ memory transfers
- memory ↔ memory transfer between machines, i.e. network communication,

as shown in Figure 1 on page 2 and Figure 4. The last item in this list is the slowest and is the focus of `HPCsim`.

(2.55) Nodes in an HPC compete for network resources and which node 'wins' (i.e. manages to perform its computation first) is dependent upon the stochasticity of the system. To see this consider the following (rather contrived) example in Figure 5, a highly simplified version of the network shown in Figure 4.

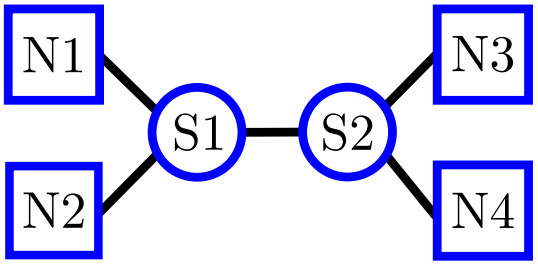

Figure 5: A network formed of four nodes {N1, N2 ,N3, N4} and two switches {S1, S2}.

(2.56) Suppose each node has one process running on it. Node N1 is performing a computation with an expected run time of 1 second and needs to communicate the results to node N3. It will take 3 seconds to do this communication. Simultaneously, node N2 is performing a computation with the same 1 second expected run time, but needs to communicate with node N4. Nodes N3 and N4 need data from N1 and N2 respectively to continue their own calculations.

(2.57) If we consider a (vastly oversimplified) model of the network in which only a single communication may be using a link at any one time, then either N1 will attempt to communicate first, blocking the N2 ↔ N4 communication for 3 seconds; or N2 will be the first node to attempt a communication, blocking the N1 ↔ N2 communication for 3 seconds (see Figure 6).

(2.58) Which of these occurs will be determined by the small differences in the time it takes N1 and N2 to complete their computations. There will also be some

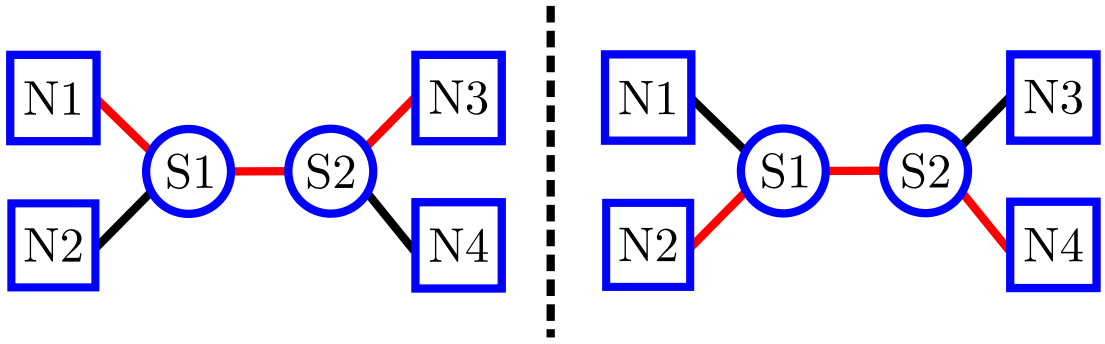

Figure 6: The two communications that need to take place, and cannot do so simultaneously.

stochasticity in the time network communications take to complete. These will add up to create a large time difference between N3 and N4 completing their computations.

(2.59) Depending on the details of the problem being solved, it might be desirable for, say, N3 to complete the computation earlier than N4, and we would design a strategy accordingly ('N1 sends first').

(2.60) Understanding how this stochasticity behaves in a large and realistic network from first principles would be a formidable task. The goal of `HPCsim` is to allow this to be studied by numerical experiment.

2.4.2 The general plan of `HPCsim`

(2.61) We want to simulate a HPC in such a way that will allow us to explore network bottlenecks. It should also allow us to run the simulation rapidly on a small machine. The code is object-oriented and written in Python 2.

(2.62) There are two main classes in the code:

- `Process` objects represent individual processes of the HPC, of which there may be several per node.
- `Switch` object represent both the switches and the shared node memory of the HPC. At present `Process` ↔ `Switch` links are identical to `Switch` ↔ `Switch` links. A more realistic model would assign much lower latencies to the former.

(2.63) The HPC is formed as a graph of objects of these classes, with `Process` objects having only a single edge connected to them. The main loop of

the program maintains a 'birds-eye' view of this graph through the lists[2] `SwitchList` and `ProcessList`.

2.4.3 Doing a computation

(2.64) We cannot simply timestep forward the wall-clock time, t_{WC}, repeatedly asking each process 'are you ready to communicate'? To do so would require us to take timesteps much smaller than the standard deviation of the time processors take to complete operations – this might even run more slowly than real-time! Instead we ask each `Process` to decide

- `tnext`, the time at which it will finish doing some computation
- `tsend`, the time at which it will next try to send anything over the network.

(2.65) The reason these two times are different is that the process might finish some computations, find that the network is busy, and so have to try again when the network is free. This is not necessarily the time at which the next batch of computation will be finished. The main loop searches through `ProcessList` for the process with the lowest value of `tnext` or `tsend`, then calls the `update()` function of that process.

(2.66) The `update()` function attempts any communication that it needs to do, and, if it is able to start doing more computation, draws a random number to determine how long the next piece of computation will take. Presently the time taken to compute each grid cell is drawn from a constant plus a lambda distribution. This gives us a minimum time to do the computation, plus a long tail.

2.4.4 Performing a communication

(2.67) The `update()` function of the `Process` object decides whether or not a communication is necessary. If it is, it calls the `mpiSend()` function of the `Switch` object to which it is attached, passing the process it wishes to communicate with as an argument.

(2.68) The `Switch` object then uses Dijkstra's algorithm to work out the fastest path to the receiving process. It then 'leases' the connection for the time it will take to complete the communication, marking each edge involved in the communication with `tlease`, the time at which those edges will once again become free. This methodology relieves us of the need to mark the edges as

[2] A computer scientist might complain that this is a dangerous procedure with any linked-list style structure, since new objects might be added to the tree without these lists being updated. We trust the main loop not to do this, since there shouldn't be a need to add new objects during the simulation anyway.

free once the communication is complete; the next attempt to communicate simply checks if the present time is later than `tlease` (see Figure 7).

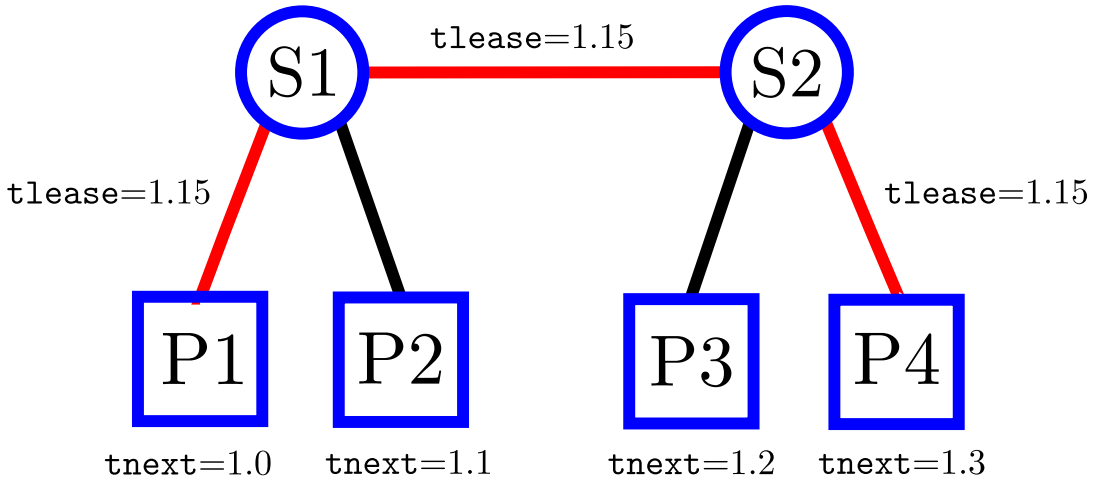

Figure 7: Process P1 completes first, and leases a connection to P4 until `tlease`= 1.15. When process P2 completes, it will have to wait until this time before it can communicate with processes P3 or P4.

2.4.5 Current state of the code

(2.69) The code still contains a number of unresolved bugs which prevent it from simulating more than a small number of computational timesteps. There are further design flaws that should be changed in the next iteration of this code. For example, the code is currently built with a one–dimensional problem in mind: each `Process` sends a single boundary value to the `Process` to its 'left', and one to the `Process` to the right. This should be generalised to higher dimensional problems.

(2.70) There is also the question of how to computationally represent the 'cone' of points that a process may compute without receiving any information from other processes. Doing this as a large array of Boolean values (representing 'have we computed this?') will slow the simulation down immensely due to the large number of memory accesses. We need a more abstract way to represent this scheme if we are to incorporate it into the simulation.

(2.71) A more realistic model of the network communication is vital and we should attempt to incorporate both the finite bandwidth and finite latency of the network. The ability of a switch to make only a finite number of connections should also be incorporated.

(2.72) The network is currently built as either a linear structure or a binary tree. This should be replaced by more realistic network topologies and random assignment of processes to processors.

(2.73) `HPCsim` will be of much more use if visualisation tools can be incorporated into it. We would like to show which timestep each process is up to, and which parts of the network are most heavily utilised.

3 Open Problems

(3.1) One of the problems that our study has highlighted is the need to better understand how switches work and how the transferring of data is performed and scales with size and competition for bandwidth. More specifically, we need to know how the time needed to copy data depends on the data volume and on the total number of transfers taking place simultaneously on the high speed network. This is important both for the analytical study and the simulations.

(3.2) There are two ways to address that problem. One would be to obtain a better description of the protocols and algorithms used by the switches manufacturers like Mirinet, Infiniband or RapidIO. This might be difficult as, unlike the Ethernet protocol, this seems to be proprietary information.

(3.3) The second method would be to do some measurements on some existing network. Ideally one would like measurements on hardware from the different manufacturers, if only to compare how they measure up against each other.

(3.4) The quantity to measure is the average time taken to copy different volumes of data, to obtain an empirical function $t(size)$ in the conditions listed below. The network should not be used for any other tasks during the measurements, as much as this is possible.

1. **Two nodes connected to a single switch without any other traffic on the switch** – $t(size)$ would provide a raw optimal performance of the switch and the protocol used.
2. **$2n$ nodes connected to the same switch without any other traffic on the switch** – this would measure how the data transport scales with the number of connections, n, on a single switch. In particular, it will be interesting to see how the system performs when the network is saturated.
3. **Two nodes connected to two different switches linked directly together or with 1, 2,... intermediate switches** – this will inevitably depend on the topology of the network. One should start with a simple topology where there is only one path between two nodes (a tree structure) and then consider topologies with multiple path like a hypercube.
4. **Two nodes connected to a multiple switch network.**

(3.5) We would like to emphasise that these measurements need only be done once to allow us to improve our model of the HPC. They do not need to be performed each time a job is submitted.

(3.6) While submitting a job in a batch queue does not allow one to know which node will be used, an administrator should be able to start them on individual nodes manually so that $t(size)$ can be measured between the two nodes. The measurements should not take more than a few seconds or minutes at the most.

(3.7) The tests described above would provide data describing how the average data transfer time scales with the volume of data, the number of simultaneous transfers on a single switch and finally how this data varies as communicating nodes are several switches away from each other.

(3.8) If only the first or the first two tests above can be performed, simulations might be able to provide reasonable data for the other, more complex, configurations. A few measurements with chained switches would nevertheless be very valuable to validate the algorithm used in any simulation.

(3.9) In this report, for the analytical estimations of the optimal partitioning we have assumed a very simple linear size dependence for the data transfer time. The constant term was linked to the latency of the network and the linear term to the average available bandwidth. The reality is likely to be more complex and exhibit a more general function. In particular, near saturation the network is likely to exhibit a more complex non-linear behaviour that simulations might be able to capture.

(3.10) More work should be done on the characterisations and numerical analysis of different PDEs (elliptical, hyperbolic, parabolic) as well as spectral methods. The robustness of algorithms to either hardware failures or errors and the scalability of the problem will depend heavily on the type of PDE being solved.

4 Conclusions

(4.1) When network bandwidth is the limiting factor, one must copy as little data as possible at a time so data should be transferred every timestep (Section 2.1.1).

(4.2) When network latency is the limiting factor, we find a value of an optimal value of k that depends on the ratio between the network latency time T_{lat} and the time needed to compute the edges of a partition (Section 2.1.2, equation (18)). This does however assume the system is homogeneous, with constant latency between nodes. In a more realistic system, one would have to solve equation (8) numerically to find the optimal value of N_p and k.

(4.3) In Section 2.3 we show that latency effects to not reduce the stability regime,

although will significantly affect problems with a large time-accuracy dependence.

(4.4) `HPCsim` detailed in Section 2.4 is in its early stages of development but could be extended to include more physically realistic properties of HPC networks, including inbuilt communication protocols and more authentic switch models. It could then be used to test scalability and robustness of different algorithms as well as different communication strategies.