Parallel Algorithms on a cluster of PCs

Ian Bush
Computational Science & Engineering Department
Daresbury Laboratory
I.J.Bush@dl.ac.uk

General Introduction To Parallel Algorithms

- In this lecture I want to cover some fairly general principles about parallel algorithm and program design. These include
  - **Task farming – don't look down upon it !**
  - **Replicated data algorithms, their advantages and their problems**
  - **Physical systems that only have short ranged interactions (in some sense)**
  - **Very basic introduction into analysing if your chosen algorithm is a good one**
  - **Distributed data – how to choose the distribution**
  - **Parallel distributed data libraries**

Parallel Computing – What Do We Want ?

- Independent operations !
  -
```
Do i = 1, n
  a( i ) = b( i ) + c( i )
End Do

dot = 0.0
Do i = 1, n
  dot = dot + a( i ) * b( i )
End Do
Call global_sum( dot )
```

- The more independent operations before comms the better (coarse grained v. fine grained parallelism)

- Strong synergy with vectorization

Task Farming (or Job Farming)

- You have N separate jobs to do, with P processors to do it on.
  - **e.g. Monte Carlo using different random number sequences**
  - **e.g. Geometry optimizations for a genetic algorithm**
  - **e.g. Loads of integrals**
  - **e.g. Studies of different trajectories when examining the scattering of ions of a material (MEIS)**
- So give each job in turn to the processors, until done
- This is task farming, and is a VERY effective way to use a parallel machine (provided you didn't spend too much on the interconnect !)
- So if it's that simple why do I mention it
  - **As usual simple ideas have some subtleties**

Why Bother to Parallelize The Code ?

- In some cases there is no, or very little coupling, between the jobs
  - **Just fire of copies of the serial code and post-process if required**
- However often some sort of analysis across the jobs has to be performed as the job goes on
  - **Could do it by clever things with grep and files and such like**
  - **But it is such a simple thing to parallelize you might as well do it that way**

Example Message Passing Code For Task Farming

```
Include ‘mpif.h’
Real, Dimension( 1:njobs ) :: answers
Integer, Dimension( 1:njobs ) :: job_list
Integer :: me, numprocs, error, i
Call mpi_init( error )
Call mpi_comm_rank( mpi_comm_world, me, error )
Call mpi_comm_size( mpi_comm_world, numprocs, error )
Call get_job_list( job_list )
Do i = 1, njobs
    If( Mod( i - 1, numprocs ) == me ) Then
            Call doit( job_list( i ), answers( i ) )
    End If
End Do
Call analyze( answers ) ! Comms required here
Call mpi_finalize( error )
```

OpenMP is analogous – just need directive for the loop

Task Farming – The Major Consideration

- What to do about I/O ?
  - **Each Processor writes to its own file ?**
  - **Processor 0 produces all the outputs at the end ?**
  - **Have some master node that copes with all the I/O – i.e. receives messages from the other procs and acts on them ?**
- For small clusters just have each processor write to its own file
  - **Much the simplest, each processor just opens a file whose name depends on the processor rank**
- May need to think again for larger systems
  - **Analysing 100s of output files can be a nightmare**

How Many Processors Can I Exploit Effectively ?

Well as many as I have jobs, duh ! NEXT !!

But What if the Jobs All Take Different Times ?

- Load Balancing can become very important
- Two real choices
  - **If njobs >> P just take your chances**
    - With any luck it will balance out
    - In general task farms work best with njobs >> P
    - This is STATIC LOAD BALANCING
  - **Alternatively use a master-slave algorithm**
    - The master doles out the jobs in turn to the slaves
    - The slave tells the master when it has finished one job, and the master then gives it a new one until there are no left
    - An example of DYNAMIC LOAD BALANCING
    - Wastes a processor …
    - Best when there are extreme deviations in job times, or though njobs is somewhat bigger than P, it's not hugely greater than P (e.g. 3-4xP)
    - The ONLY excuse for mpi_any_source in a mpi_recv when using MPI

But My Case Will Not Fit on a Single Processor !

- You now have to move to distributed data land (see later)
- So now you MUST use some parallel programming paradigm
- This is an ideal job for MPI communicators. Use MPI_SPLIT to create communicators for subsets of all the procs in use, then use the same method as above but with these subsets instead of single processors
- This is the method use in the genetic algorithm mentioned above

I Have a Task Farm, But my Job does Other Stuff as well

- Quite a common case
- Many parallel codes use a mixture of parallelization techniques and algorithms, the one chosen is that best to perform the operation required
- The same principles apply, and as task farming is such an effective way of parallel computing one should look for even very simple opportunities if available
  - **e.g. in a UHF calculation you have to do two independent diagonalizations. Use a task farm with just two jobs, half the procs doing one diag, the other half doing the other.**
    - This is also part of the genetic algorithm calculation. Farms within Farms !

Replicated v. Distributed Data

- Replicated data is when each processor holds all the data required to perform the calculation

- Distributed Data is when you split up all the (large) arrays so that each processor only holds a part of it

- Hybrid schemes are possible

Replicated Data, Pros and Cons

- It's easier to program
  - **All the data is there, you don't have to mess about message passing to get what you want**
  - **I/O is easier, you have the whole array so just read in at the beginning and write it all out at the end**
- Much less complicate message passing is required, typically just a few collectives
- However it is memory hungry
- And typically it scales less well than distributed data

DL_POLY2 – Replicated Data and Scaling

- DL_POLY is a classical MD program out of Daresbury by W.Smith
- The replicated data version is DL_POLY2
- The major compute time is the force evaluation which is perfectly parallel and linear scaling (assuming only short ranged forces)
  - **T(compute) = aN/P**
- The major comms time is in global collectives
  - **T(comms)=bNxlog(P)**
  - **N.B. Grows with P**
- So T(compute):T(comms)=a/(bPxlog(P))
- Want this ratio to be big !

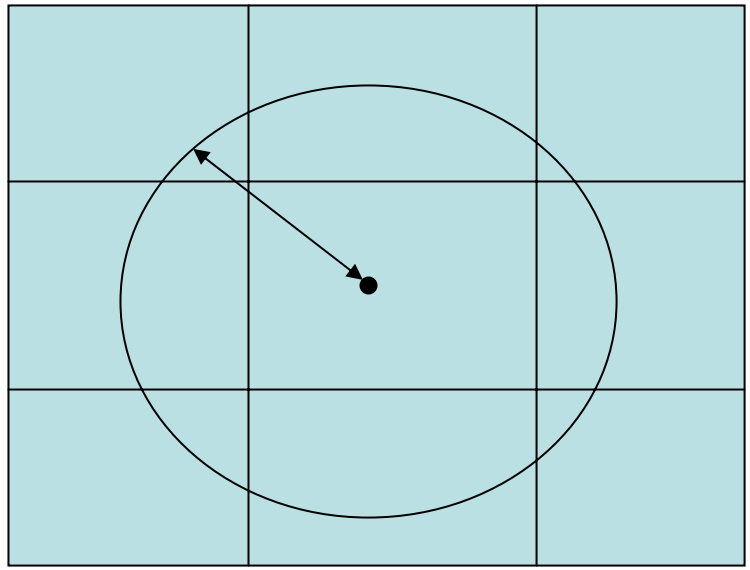
Short Ranged Interactions

- An important general class of simulations/calculations have interactions between `particles' which are in some sense short ranged – typically when compared to the size of the bounding box

- Examples might be
  - **Classical MD simulations where there are no Ewald terms, e.g. Lennard-Jonesium**
  - **Many Grid based simulations – often solving PDEs**
    - CFD
    - Oceanography
    - Weather forecasting
    - Climate modelling
- What happens if we can give each processor a bit of the box which is larger than the range of the interaction ? – i.e. distributed data
- Domain Decomposition

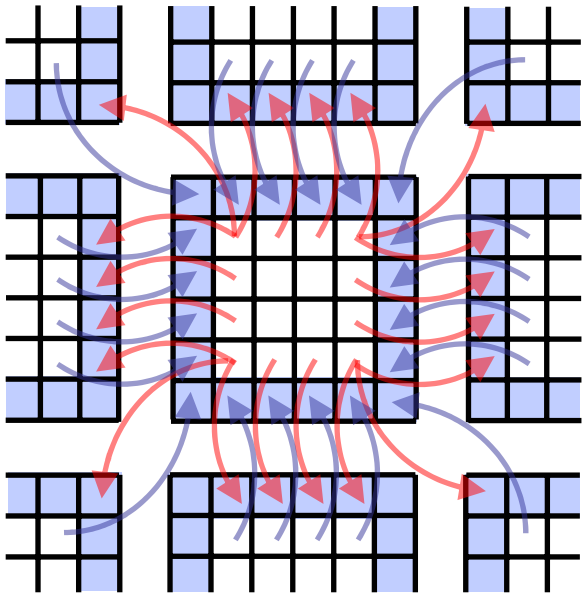
DL_POLY 3

- DL_POLY3 is a distributed memory parallel classical MD code

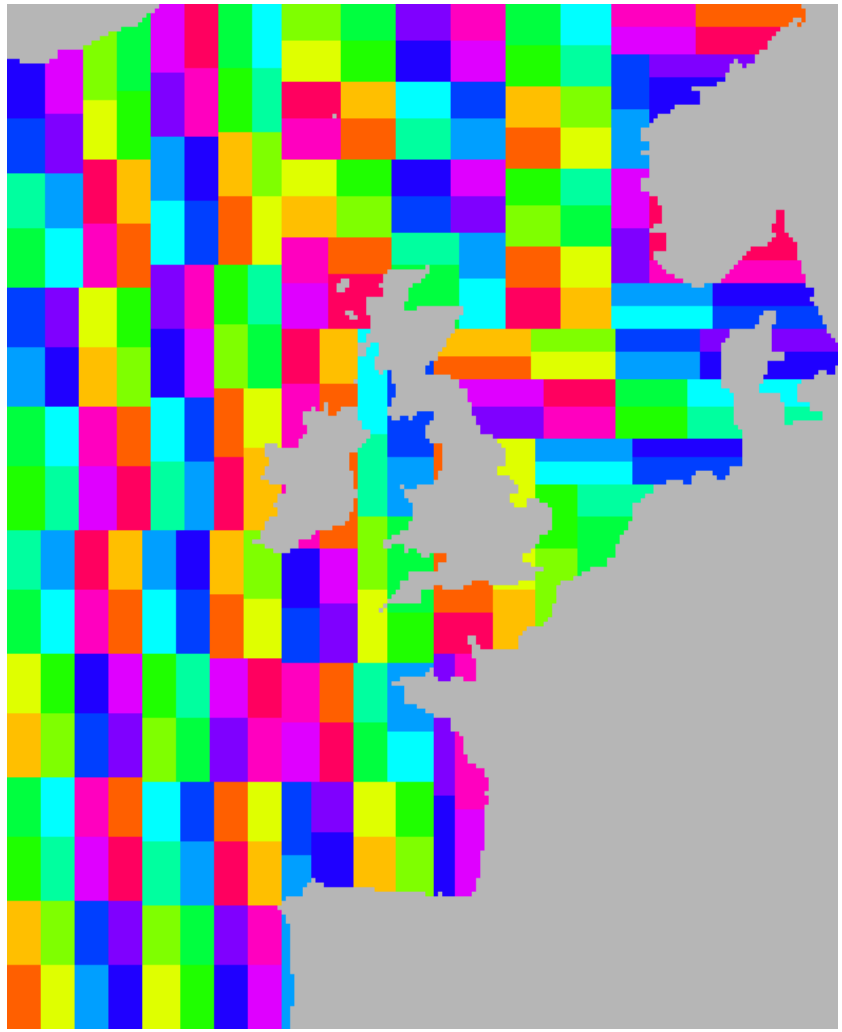


Only need to communicate with nearest processors !

Recursive bisection grid partitioning in POLCOMS



2D halo boundary
data exchange



Scaling With Short Range Forces

- Use DL_POLY as an example – POLCOMS similar but a little more complicated

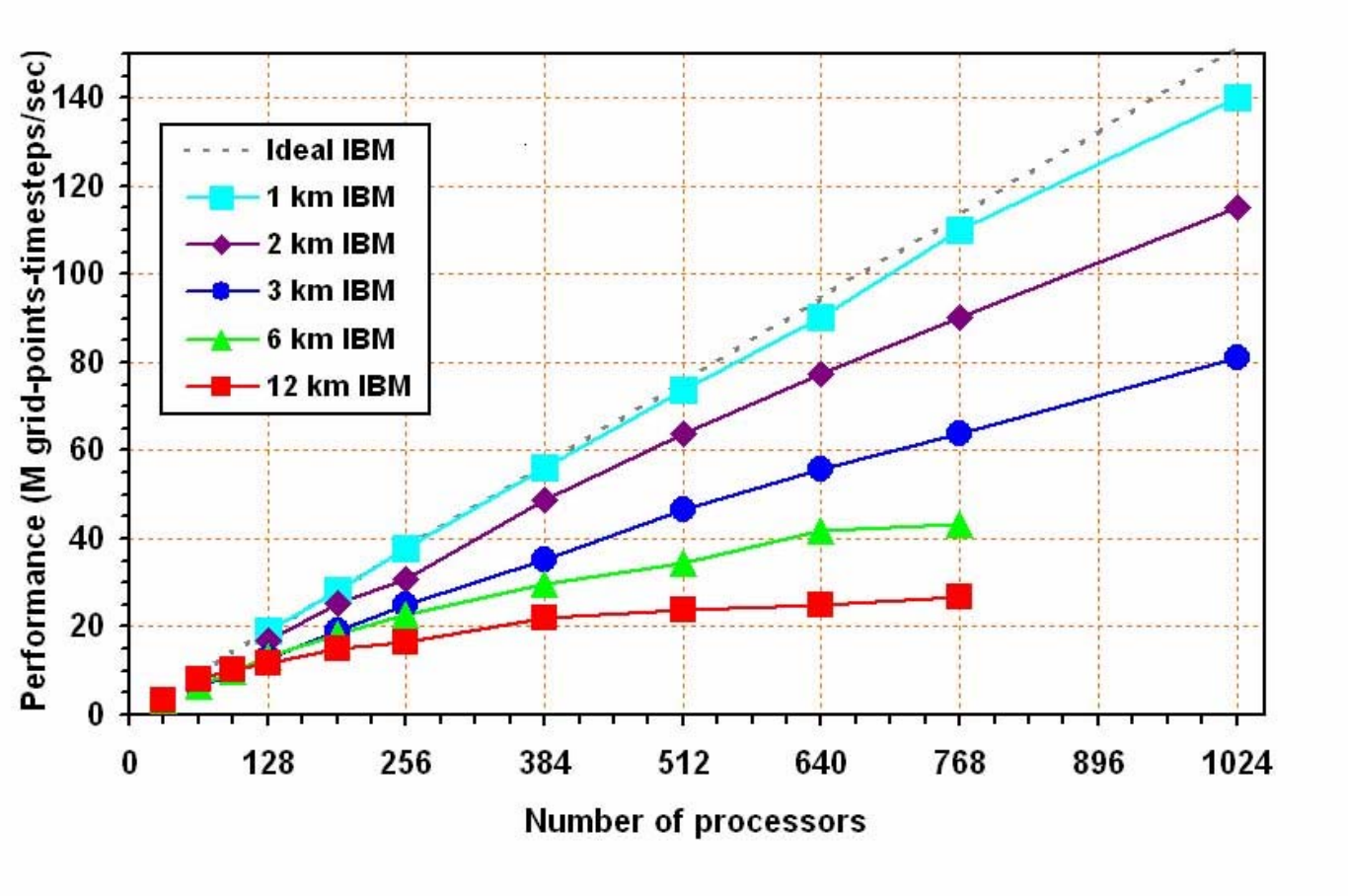
- Remember for the replicated date DL_POLY2
  - **T(compute) = aN/P**
  - **T(comms) = bNxlog(P)**
  - **T(compute):T(comms) = a/(bPxlog(P))**

- For distributed data DL_POLY3
  - **T(compute) = dN/P - note new pre-factor**
  - **T(comms) = eN/P - Each proc has N/P data and have to communicate with a finite number**
  - **T(compute):T(comms) = d / e**

- INDEPENDENT OF PROCESS COUNT and system size

- So should scale perfectly …

So Why Isn't POLCOMS Perfect For All System Sizes ?



The Effect of Latency

- Possible reason include
  - **Load imbalance**
  - **Amdahl's law effects**

- But we haven't considered the effect of latency in the communications time:
  - **T(compute) = dN/P** **- still, no latency effects**
  - **T(comms) = f(α+βN/P)** **- residual comms time you can't get rid of**
  - **T(compute):T(comms) = dN/(f αP) + d/(f β)**
    = (d/f)(N/(αP) + 1/β)

- Latency introduces a term that DECREASES the ratio as P increases
  - **Bad !**

- So under what conditions do we expect good scaling ?

A Closer Look At The Ratio

- T(compute):T(comms) = d/f x (N/(αP) + 1/β)
- To get a ratio that does not decrease markedly with processor count we want a small latency
- To get a large ratio overall we want
  - **Large Bandwidth - 1/β**
  - **Each bit of compute work being expensive – d**
    - Another way of looking at it is Volume to surface area effects
    - Or have slow CPUs !!
  - **Small numbers of messages – f**
  - **LARGE SYSTEM SIZE – big N**

Parallel Computing Is Best For Large Systems

Distributed Data

- For small clusters replicated data can get you a long way.
- However as your problems become more complex, and as you gain access to machines with more processors, Distributed Data algorithms become more and more attractive.
- It's important to understand that different versions of the same parallel program which use different data distributions are often best suited to different problem domains
  - **DL_POLY2 – replicated data – a few 10's of thousand atoms, a few 10s of processors**
  - **DL_POLY3 – distributed data – 100's of thousands of atoms**
- However it is using distributed data that I find most interesting
  - **More intellectually challenging**
  - **Allows much larger and more difficult problems to be solved**

Distributed Data – How To Distribute Your Data

- So you have a program, or at the very least an algorithm, to solve your problem, and you think that distributed data is for you. How should you choose how to split up your large arrays ?

- Firstly look at what people have done in related areas before
  - **They might have a solution ready for you**
    - The programmer's first commandment is 'Be Lazy' – don't reinvent the wheel!
  - **If there is not a ready made solution, previous work will suggest how to distribute your data**
  - **Be imaginative in your reading !**
    - As shown above there is some relationship between classical MD and ocean modelling
  - **Be aware of common distributions, e.g. domain decomposition (or blocked), row, column, block cyclic (see later) …**
    - They are used for a reason

Distributed Data – How To Distribute Your Data (2)

- So you can't find anything, or at maybe what you have found is not very convincing.
- Now you have to think ….
- You have to
  - **Understand your algorithm for the whole calculation**
    - How does the time scale with system size
    - How does the memory scale with system size
  - **From that which parts take the most time and/or memory**
    - If possible PROFILE !
  - **Understand the data flow in your program**
    - For a good parallelization this is possibly the most important

Distributed Data – How To Distribute Your Data (3)

- The way to choose your data distribution is to base it on that which best fits the most expensive part of your algorithm (in terms of time or memory, whichever is your major concern)
  - **Fairly obvious really**
  - **The distribution of all other object within your code should derive from this**
    - Avoid unnecessary comms
  - **Is there a library that does what you want ? (see below)**

- Sometimes more than one part of the algorithm is expensive
  - **If you are lucky the same data decomposition can be used for both, e.g. for dense linear algebra in general block cyclic is a very good choice**
  - **If not you either have to redistribute between the expensive parts …**
    - How expensive is the redistribution relative to the other parts of the calculation ?
  - **Or compromise**

Distributed Data – Is It Worth It ?

- Before you jump in at least have a rough and ready estimate of the compute:comms ratio for what you are thinking about. Remember you want that high, not to decrease markedly with number of processors, and at the very least be constant with system size. Include latency if possible: e.g.
  - **Matrix Multiply - N^3 compute, N^2 communicate**
    - Very good. Good ratio and large messages
  - **Matrix Vector Multiply - N^2 compute, N communicate.**
    - Good. Good ratio but messages getting a bit short – latency effects ?
  - **FFT - Nxlog(N) compute, at least N communicate**
    - Hmmmm, poor ratio, short messages. 1D is a bit of a disaster but higher dimensionalties work better, see below
  - **Matrix transpose - 0 (ZERO) compute, N^2 communicate**
    - **COMPLETE DISASTER !!!!**

Rules of Thumb For Good Algorithms

- Algorithms that scale very well generally have two properties
  - **A good compute to communicate ratio**
  - **Only local comms are required, i.e. to perform the operation a given processor does not need to know the state of the WHOLE machine, and best the amount of processors it needs to communicate with should grow less slowly than the total number of processors**
    - e.g. Matrix diagonalization has a good ratio but is difficult to get to scale well – orthonormailty constraints
- The first is required, the second is not but should be held in mind

Parallel Libraries

- Remember the first commandment – 'Be Lazy'
- There are increasingly large number of good quality libraries that solve a wide variety of problems in parallel already out there. If they
  - **Fit your data distribution for your whole code**
  - **Or redistribution is a possibility**

  you should use them.

- Ideally the library suits all your needs, at least computational one, and then your data distribution should be driven entirely of that of the library.

- I would strongly suggest you stick with open source libraries
  - **Portability**

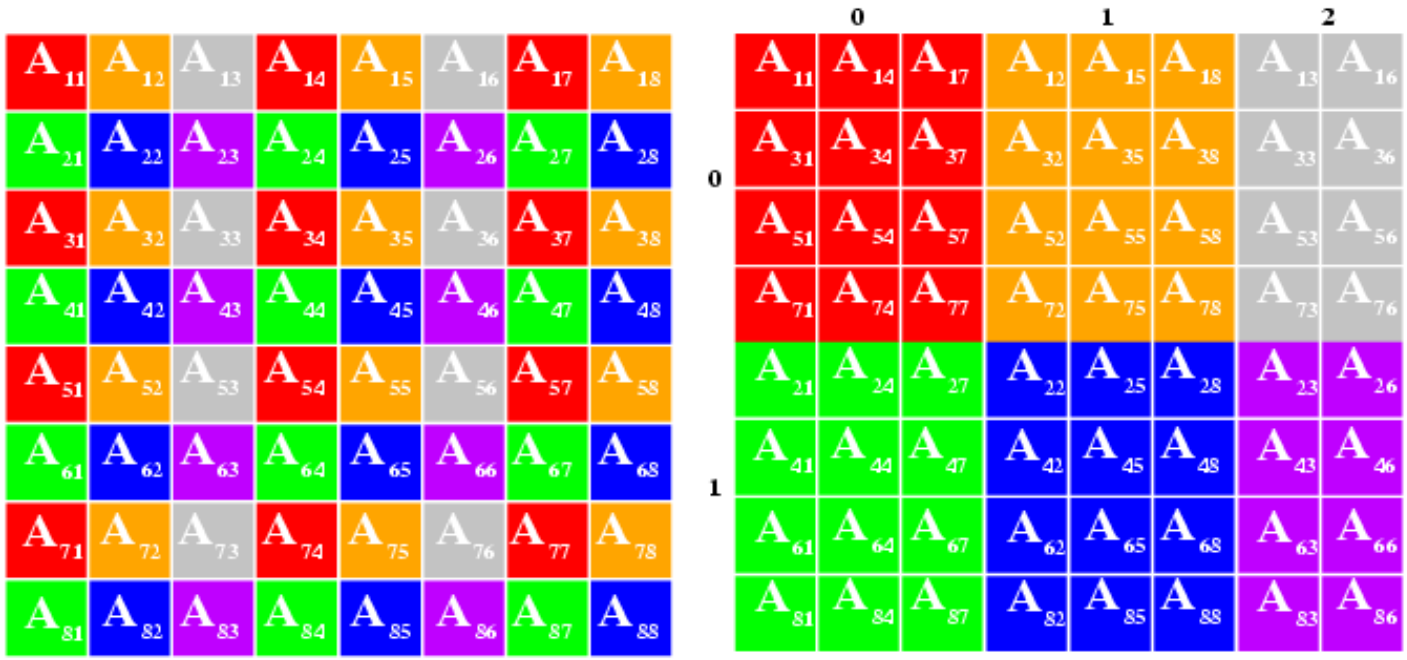
One or Two Example Parallel Libraries

- There are loads out there. Some that myself and my collaborators have use are
  - **ScaLAPACK - Dense Linear Algebra**
  - **PLAPACK - Dense Linear Algebra**
  - **FFTW - Multidimensional Parallel FFTs**
  - **PetSc - Partial Differential Equations**
  - **ParPACK – Sparse Eigen systems**
- Remember – DON'T REINVENT THE WHEEL !

One Example - ScaLAPACK

- Parallel version of LAPACK
  - **Parallel BLAS**
  - **Linear Equation Solves**
  - **Standard Matrix Factorizations**
  - **Eigensystems**
  - **Singular Value Decomposition**
  - **Linear Least Squares**
  - **…**
- I’ve used this extensively and it is a very good library (with one or two reservations about the eiegnsolvers) – CRYSTAL, GAMESS-UK, PFARM
- Uses a block cyclic decomposition

2-D Block Cyclic Distribution



Global (left) and distributed (right) views of matrix

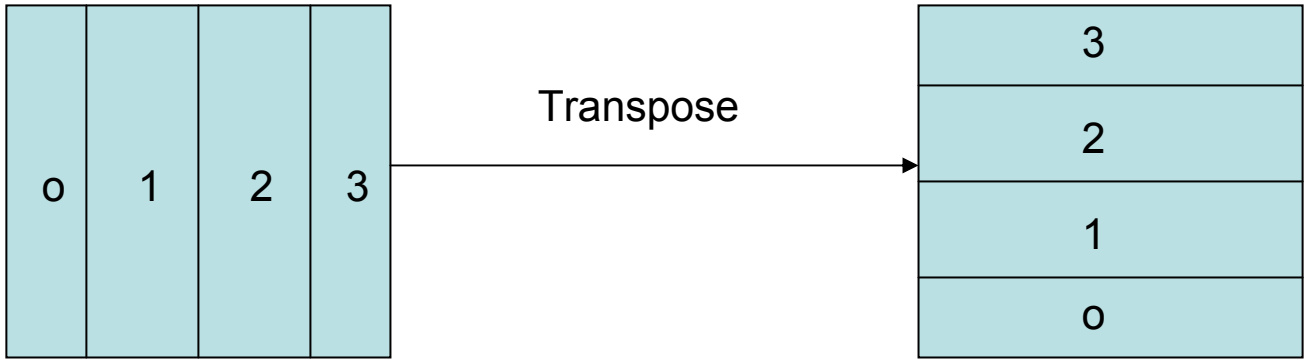
- **Flexible**
- **Load balancing and scaling**

But Sometimes Life Just Gets Difficult

- Sometimes the compute to comms ratio is poor, it's a global operation, and you've got to parallelise it because it takes too much time, or too much memory, or both.

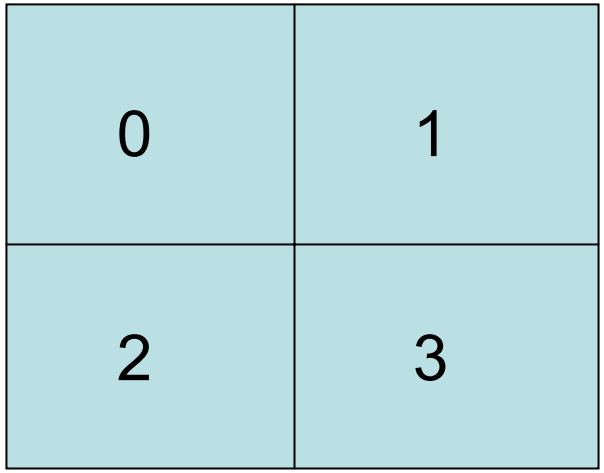
- Example – Three Dimensional FFT - Used huge number of areas
  - **Plane wave *ab initio* electronic structure codes, e.g. CASTEP, CPMD, VASP**
    - In parallel the FFTs often take over 50% of the run time
  - **Ewald Summations, e.g. DL_POLY3**
    - Remember the earlier stuff was for short range forces only
  - **Spectral Methods in CFD**
  - **Signal Processing**
  - **…**

2D FFTs – Method 1 – Transpose Method



- Do the FFTs in the Y Direction in parallel using a serial library routine
- Transpose – Requires all to all communication
- Do the FFTs in the X direction in parallel using a serial library routine
- T(compute) = aAlog(a) – where A is the area, i.e. number of grid points
- T(comms) = bP(α+A/βP^2) – latency problems
- Used by CASTEP, VASP, CPMD etc.

2D FFTs – Method 2 – DaFT Method

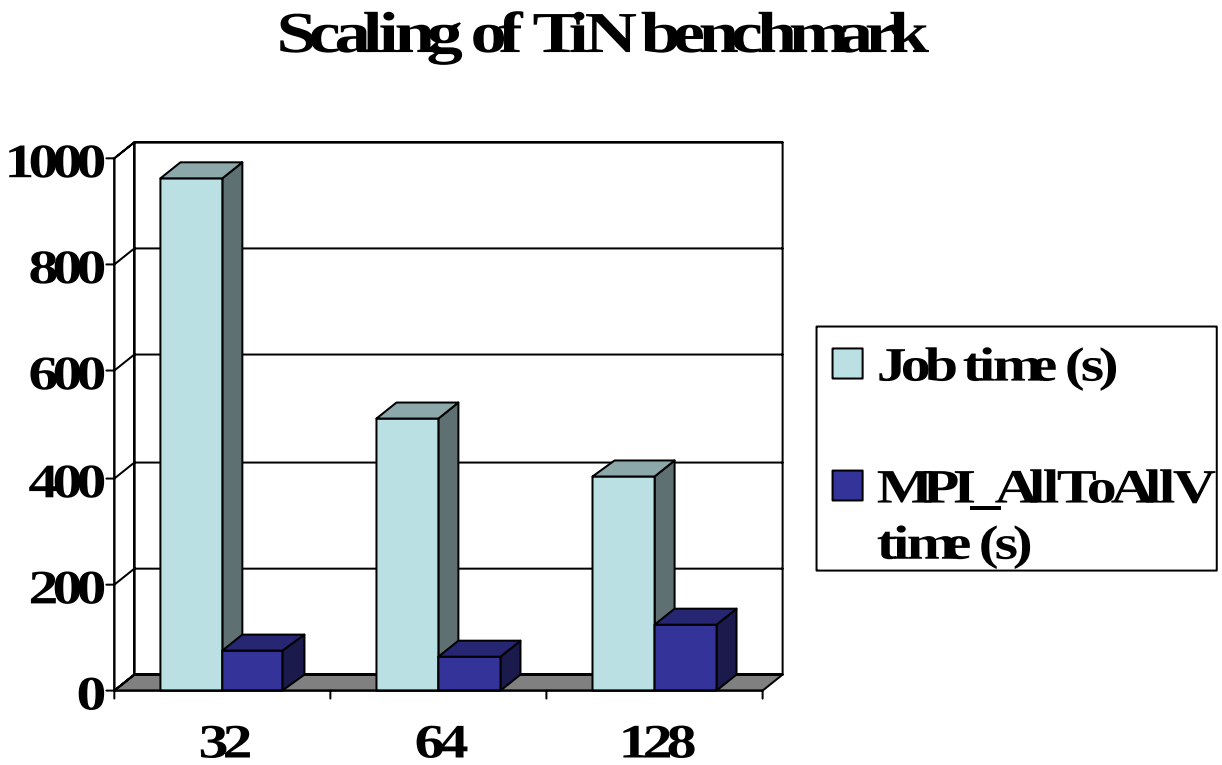


- Do the FFTs in PARALLEL in the X direction – lots at once
- Do the FFTs in PARALLEL in the Y direction – lots at once
- T(compute) = cAlog(A)
- T(comms) = dlog(P)(α+A/βP)

Comparison of the Two Methods

- Transpose Method:
  - **T(comms) = bP(α+A/βP^2)**
- DaFT Method:
  - **T(comms) = dlog(P)(α+A/βP)**
- DaFT passes more data about but the messages are much longer
  - **In the latency limit DaFT is better**
  - **In the bandwidth dominated regime Transpose is better**
- So
  - **Transpose better for small P**
  - **DaFT better in the very large P limit**
- In practice Transpose is generally better, at least on most machines
- So why does DL_POLY3 use DaFT ?
  - **One for you to think about – see next lecture !!**

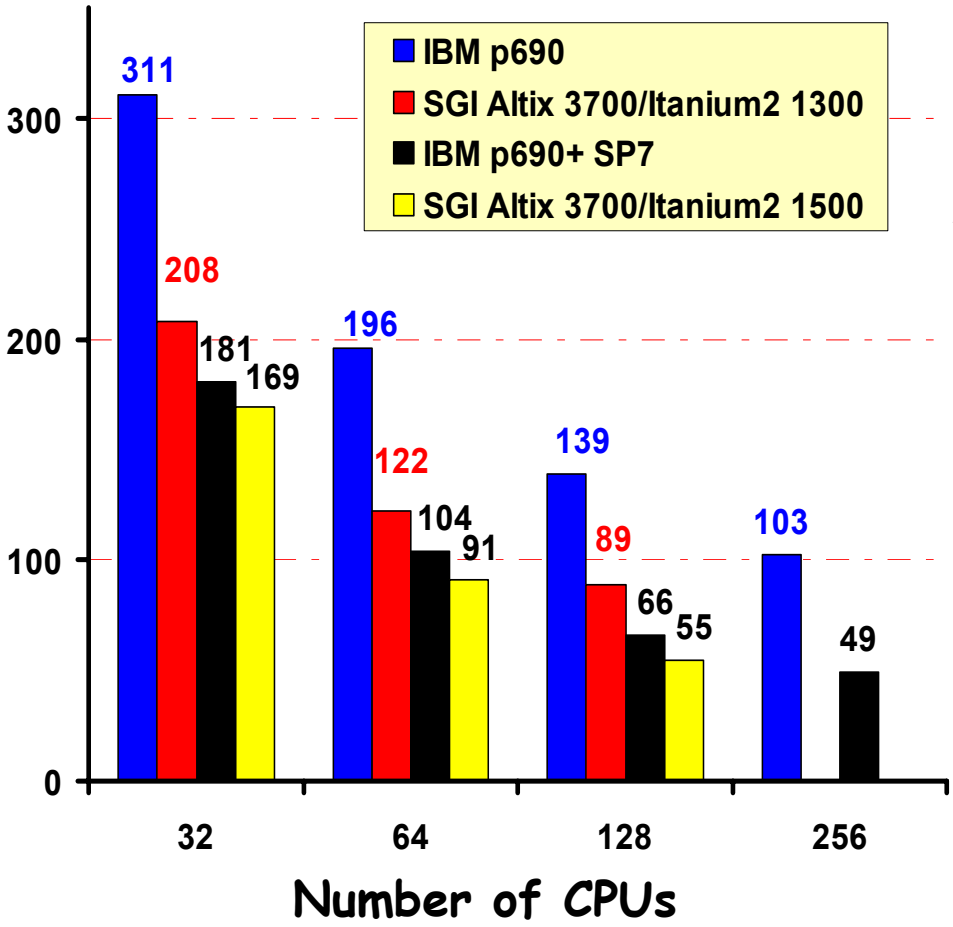
Castep on HPCx



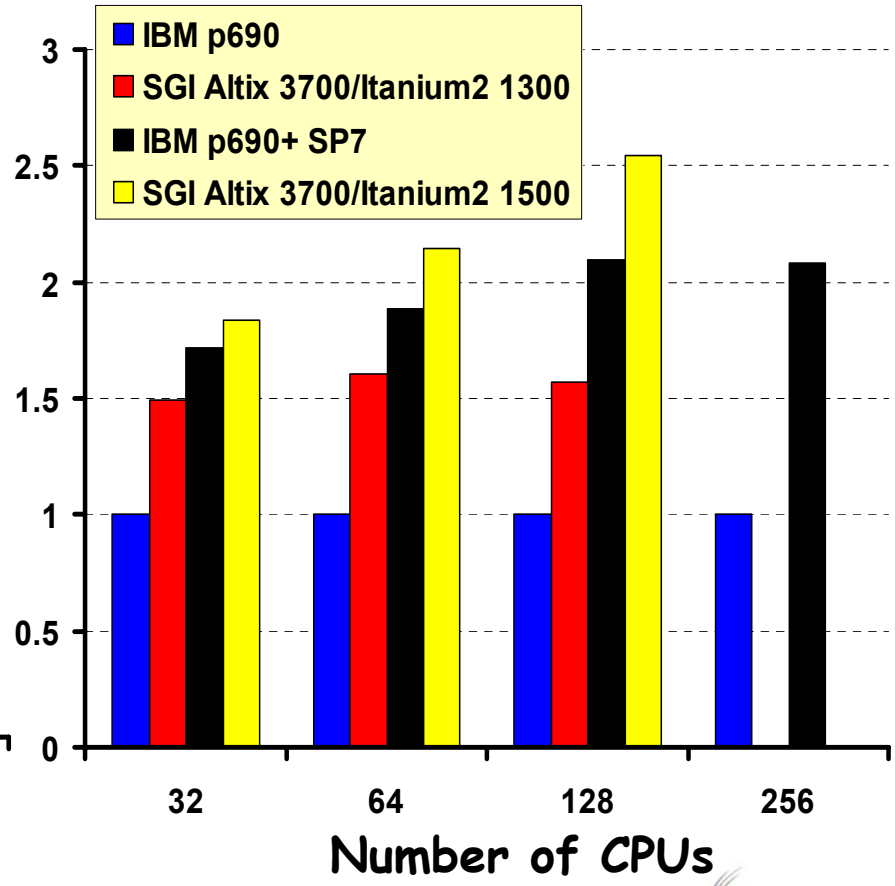
DL_POLY3

Gramicidin in water;
rigid bonds + SHAKE:
792,960 ions, 50 time steps

Measured Time (seconds)



Performance Relative to the IBM p690



Summary

- Parallel computing is all about finding independent operations
- Task farms are an easy and effective way to use parallel computers
- Replicated data is a comparatively simple way to program the beasts, but has memory and scaling drawbacks
- If your interactions are short ranged you should get good scaling
- Try to think about the compute:comms ratio
- Distributed memory requires good understanding of the algorithm
  - **Base your distribution on the most important parts**
- Don’t reinvent the wheel – Use google, Use those libraries !
- What’s good for small processor counts may not be good for large