MPI Applications Course Overview

- Introduction
- Basic performance concepts
- MPI performance issues
- Tools
- Parallel Decomposition
- Example Applications

Message Passing Application References

- References from the APAC MPI Programming Course for general MPI syntax etc.
- Various course documents from EPCC. See particularly the Decomposing the Potentially Parallel course (html, PDF and slides).
- Designing and Building Parallel Programs by Ian Foster - an excellent online book for parallel application development
- Domain Decomposition course from Stuttgart HPC Center

Introduction

Course is

- motivated by common user problems and misconceptions
- not specific to MPI - much is applicable to any message passing and even to shared memory parallel (eg. OpenMP) computing.

Parallel computing (i.e. parallel application development) is:

- not easy
- not always applicable or necessary

Object of the Exercise

To use multiple cpus (or greater resources) to *significantly* decrease the *time* a particular computational task takes to complete.

Pre-Parallel

Before going parallel:

- understand and sort out general issues of single cpu performance first (memory and cache use, IO, ...)
- use a good (fast) algorithm for the serial code
- develop a working serial (at least prototype) code first
- take James Murray's advice* - steal a parallel code!

* obscure reference to an APAC Summer School presenter

Measuring performance: CPU Usage?

- High %CPU usage is almost meaningless!
- On AC, both MPI and OpenMP may "spin wait", i.e. when waiting (for messages etc), processes "spin" in a tight loop constantly requesting a response.*
  Why? Because it gives the best communication performance.
- Leads to 100% cpu usage doing nothing.
- Low %CPU usage indicates a problem (often IO)

* has implications for master-slave codes

Exercise 0: cputime

The following code only has one active process at any time - can you see why?

Compile (`ifort exer0.f90 -o exer0 -lmpi`) and run with

```
> qsub -q express -lncpus=4,walltime=0:02:00,vmem=450mb -wd
 mpirun ./exer0
 ctrl-d
```

and check the cpu usage of all processes using `qps` *`jobid`*. (Redo this exercise with Vampir later.)

```
program exer0
  implicit none
  include 'mpif.h'
  integer :: mpirank, mpisize, err, i, j
  real(8) :: s

   call MPI_Init(err)
   call MPI_Comm_rank(MPI_COMM_WORLD, mpirank, err)
   call MPI_Comm_size(MPI_COMM_WORLD, mpisize, err)
   s = 0.0
   do i=0, mpisize-1
      if (i == mpirank) then
         do j=1, 50000000
            s = s + log(sqrt(1.1+cos(0.00001*j+mpirank)))
         end do
         write(*,*) s
      end if
      call MPI_Barrier(MPI_COMM_WORLD, err)
   end do

   call MPI_Finalize(err)
 end program exer0
```

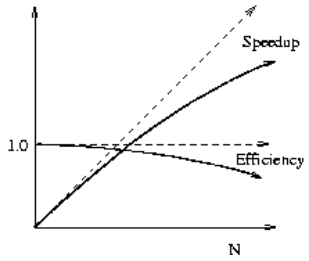
Dedicated CPUs and Walltime

- Timesharing cpus a huge waste of time
  (spin waiting, 10us process synchronisation and 10ms process timeslices => most time spent in spin wait)
- Jobs given dedicated access to allocated cpus during job.
- Only sensible measure of parallel jobs is **walltime**

Measuring Performance

```
                      walltime(1 cpu)
Speedup(N cpus) = ----------------
                      walltime(N cpus)

Efficiency(N cpus)=Speedup(N cpus)/N
```



Amdahl's Law

The parallel speedup of your code is limited by the unparallelised part of your code (the single cpu or redundantly executed parts).

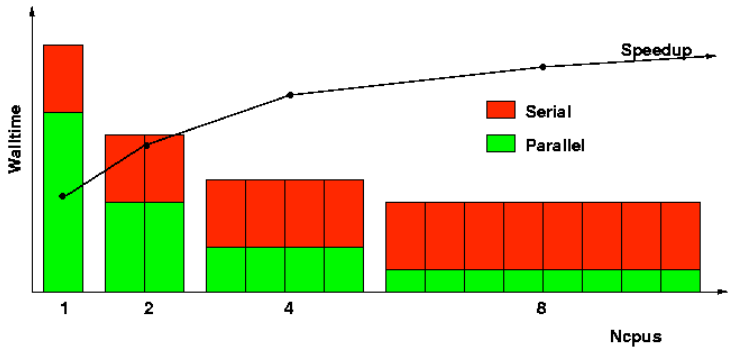
If

- `N` is the number of processes,
- `S` is the fraction of operations which are serial and
- `P (=1-S)` is the fraction of operations which are parallelized (distributed amongst cpus),

then

```
                                1
Speedup(N cpus) = ---------   ->  1/S  as  N increases
                  S + P/N
```

Amdahl's Law: Theory



Sequential Components

- Code executed by a single process:

```
if (rank == 0) { ... }
```

- IO though a single process
- Operations done redundantly on local copies of redundant data

Exercise: Amdahl's law

The script `amdahl` runs the 2D finite difference code `overlap.f90` on a 192x192 grid using 1,2,..,64 cpus. Compile `overlap.f90` to produce an executable called `overlap` (use -O3) and `qsub amdahl` to investigate the speedup of `overlap.f90`.

Use

```
grep Walltime amdahl.o#####
```

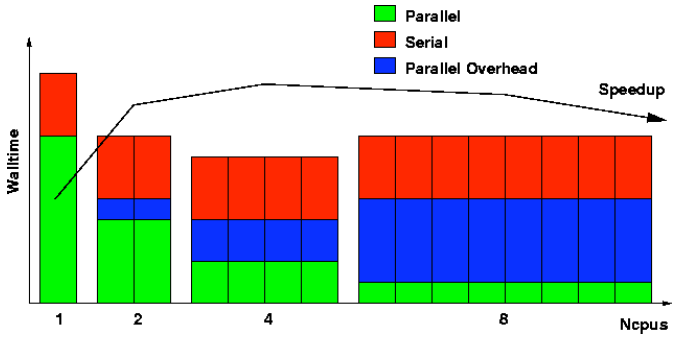
to see the scaling. Note that "Cost" is ncpus*walltime which is proportional to the real job cost.

Run

```
./speedup_efficiency.sh amdahl.o#####
```

to see how Amdahl's Law holds up.

Amdahl's Law: Practice



Parallel Overhead

- Any operations not in the serial code
- Number of such operations often increases with N cpus
- Any message passing or synchronization (MPI_Barrier)
- Extra redundant computation
- Algorithm changes
- Extra system overhead, e.g. forking threads for `parallel do/for` in OpenMP

Problem Size

Often CANNOT use parallelism to solve a small problem faster but usually CAN solve bigger problems.

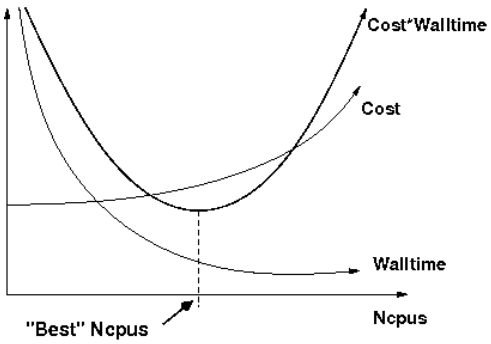
Dedicated or Shared machine

- What is "good" speedup depends on the system you use.
- If you have access to an idle parallel system (your own maybe!), any speedup is OK
- APAC-NF is a heavily subscribed system used by 100's of researchers - any excessive wastage due to parallelism effectively decreases everyone's usage.
- Make the most of your grant - dont waste it on inefficient parallelism.

Yeah, but how many cpus?!

Want to minimize both:
1. time to solution (walltime)
2. cost (= ncpus*walltime)
Contradictory!

How about minimizing Cost*Walltime?



Exercise: Choosing ncpus

Edit the script `amdahl` so that the arguments to each mpirun line are "768 1000" (a larger problem, less timesteps) and submit the script again. Repeat the `amdahl` exercise but this time also run

```
./select_ncpus.sh amdahl.o#####
```

What is the suggested number of cpus to use?

Scalability

- *Strong scaling*: speedup solving a fixed size problem distributed over more cpus.
- *Weak scaling*: speedup solving a fixed size problem *per cpu* using more cpus.
- *Memory scalability*: ability to recruit more cpus to get more memory to solve a larger problem.

IO - the good, the bad and the ugly ...

Most of it is UGLY!

- Filesystems on large parallel systems geared to BIG IO - cannot support 500 "general purpose" users.
- What works OK on your PC may be disastrous on the AC
- Do IO in big chunks as infrequently as possible (like messages - see below)
- Filesystems for all occasions ...
- Talk to us about what your best options are.

Exercise: IO

Compile and run each of:

- `poorio.f90` (no, dont do this one!),
- `goodio1.f90`,
- `goodio2.f90`,
- `goodio3.f90` ,
- `bestio.f90`

and compare the different IO methods. Beware, the array sizes are different and the timings include the computation (which should be negligible).

Important: delete all large files!

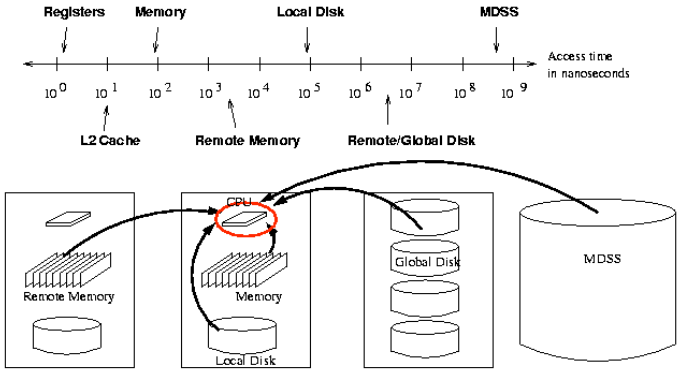
Parallel IO

DON'T have multiple processes write to the same file (except in MPI IO)! Garbage results since data streams conflict but worse, its VERY slow.

Options for IO from parallel jobs:

- Use a nominated IO process for all IO -
  - maintains serial IO behaviour
  - independent of ncpus
  - limits scalability
- Use MPI IO -
  - independent of ncpus
  - maintains serial IO behaviour
  - may not be great performance
- Each process has its own file -
  - best performance
  - requires conversion modules/utilities to vary ncpus

Disk IO Time Scales and Locality



MPI Issues

Granularity

fine grained

- fundamental compute tasks (between synchronization or communication) are small
- frequent interactions between data in different processes
- reflects "tightly coupled" problem

coarse grained

- infrequent interactions between data in different processes
- fundamental compute tasks are large
- reflects "loosely coupled" or uncoupled problems

Granularity

Aim: To make a fine grained problem as coarse grained as possible!

Possible methods:

- increasing data locality to avoid communication
- reordering operations
- decoupling tasks using the right MPI routines

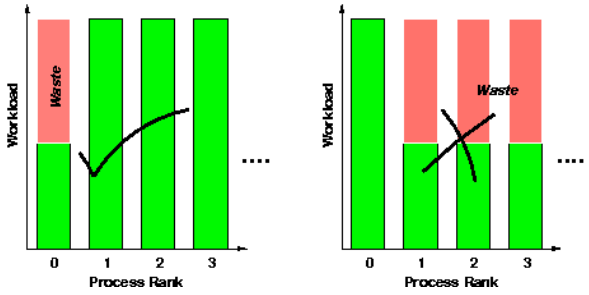
Scalable Parallel Computing

Three Golden Rules of Scalable Parallel Computing:

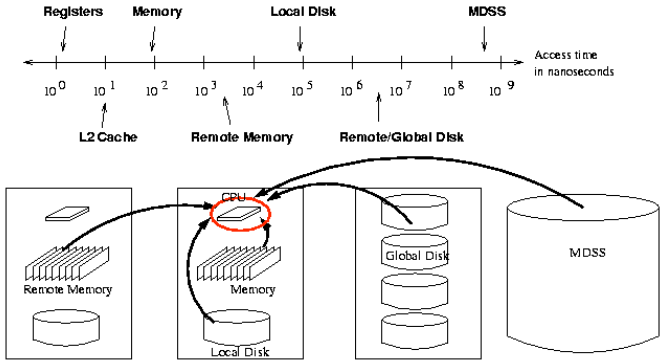
- Load balance
- Use a coarse grained algorithm
- Load balance

Load imbalance

- Message passing time is NOT usually the problem.
- Main problem is having processes waiting for other processes (usually sitting in MPI calls)
- If you must have a load imbalance, have a small number of processes with less work than the majority, not more.



Message Passing - Time Scales and Locality



Message Passing Performance

Exercise: Compile the code

```
> cc pingpong.c -o pingpong -lmpi
```

Run pingpong with two processes:

```
> qsub -q express -lwalltime=0:30,ncpus=2,vmem=500mb -wd
mpirun ./pingpong
ctrl-d
```

Two things to note:

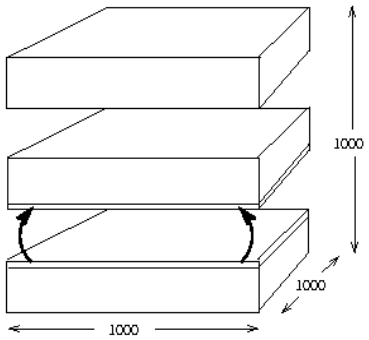
1. Shortest message take 1us or greater:

   Any message costs at least 1600 floating point ops.

2. Sending 1000 messages of 64B takes 2.8ms
   Sending 1 message of 64KB takes 0.26ms

   Try to use larger messages.

Communication



Practical example: Divide a 1000x1000x1000 64bit array over the last dimension. The cost of sending one 1000x1000 slice from one processor to another is about 0.025 seconds (8MB/320MB/s) or 25 million flops (assuming 1Gflop/cpu). If communication is to be 10% of computation then

- if flops are `a(i,j,k) = a(i,j,k)*a(i,j,k-1)` can only use 4 cpus well
- if flops are `a(i,j,k) = sqrt(a(i,j,k-1))` can use maybe 40 cpus

Scaling of communication

- Theoretical discussions often present scaling of computation vs communication
- 3D finite difference solver has O(N^3) operations and communication volume O(N^2) => "good scaling"
- Parallel 3D FFT solution involves O(N^3*ln(N)) operations and O(N^3) communication volume => "not so good scaling"
- Talking about scaling **problem size** not scaling number of cpus!

Types of communication

Blocking, nonblocking, synchronous, asynchronous, yadda, yadda, ...

- Generally choose type of point-to-point communication for overall algorithm behaviour and load-balance
- not for performance of any one message pass

Types of communication

Examples:

Identical workload on each process, processes execute in lockstep, natural frequent synchronization	=> blocking synchronous
Unknown adaptive communication pattern, small data volume	=> buffered (asynchronous)
Unknown adaptive communication pattern, larger data volume	=> non-blocking synchronous

Synchronous

- Often fastest type. Directly from process buffer to process buffer - no extra copying or buffering of data.
- No reliance on user-provided or system buffer space.
- Blocking synchronous simple and effective for simple distributed grid jobs.

Asynchronous

- Useful for highly unstructured code (processes not naturally communicating at the same instant - different task lengths etc).
- Avoids having to manage outstanding requests (cf non-blocking)
- Do have to mange buffer space
- Can be slower - data is copied into and out of buffers.

Blocking

- Want to be able to use/reuse buffer immediately
- Dont have a "window of time" in which to send/receive message
- Easy to implement and often used at the other side of a non-blocking call. (Must know this call will be second in time).

Non-blocking

- Way to avoid deadlock.
- Another way to handle unstructured adaptive communication
- Need to manage outstanding requests
- May be able to overlap communication with computation, hence your communication cost (almost) disappear (still have to manage requests).

Non-blocking 2

- Initiate the non-blocking request as early as possible (as soon as buffer can be used for message)
- Make the completion call as late as possible (just before buffer is needed for computation)
- Makes the "communication window" as large as possible
- Often only need one side non-blocking

Exercise: MPI point-to-point types

Implementing `MPI_Alltoallv()` - all processes send to all other processes and the size of the message sent to each process varies.

Compare

- **pairwise blocking exchange:** have to carefully coordinate processes to communicate in pairs of processes and make sure all pairs are done. Within pair exchange, avoid deadlock!
- **concurrent non-blocking:** just throw all the messages out their simultaneously and let MPI sort them out.

```
ac:~ > icc alltoallv1.c -lmpi -o alltoallv1
ac:~ > icc alltoallv3.c -lmpi -o alltoallv3
ac:~ > qsub -lncpus=8,vmem=8gb,walltime=10:00 -I –wd
...
> mpirun ./alltoallv1 1000   # argument is largest message size
...
> mpirun ./alltoallv1 10000
...
> mpirun ./alltoallv3 1000
...
> mpirun ./alltoallv3 10000
...
> exit
ac:~ >
```

Try to understand the non-blocking version. Run these through Vampir later.

SGI MPI and compiler flags

The SGI MPI library is a dynamic library (has to be!).
The Intel compiler option `-fast` is the same as `-O3 -ipo -static`
so cannot be used. Use `-O3 -ipo` instead.

Tools

Various tools are provided at the National Facility for debugging and profiling parallel codes.

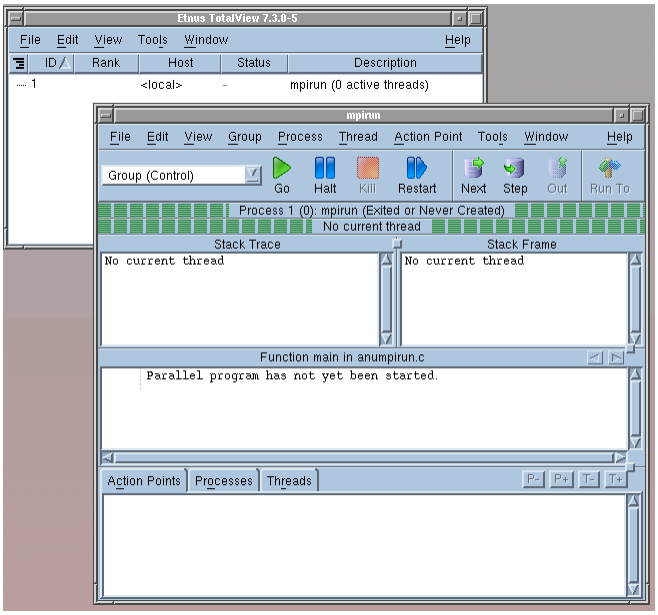
Totalview

Totalview is a graphical debugger allowing you to step through your parallel code.
You can step each process through inidividually or as a group. You can see what
messages are outstanding etc...

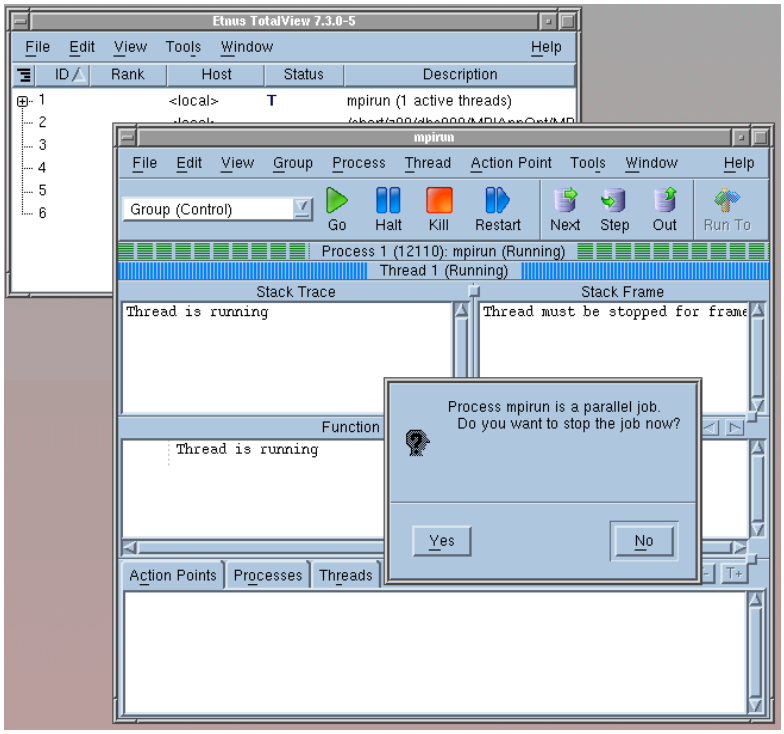
Usage:

```
> module load totalview
> ifort -g overlap.f90 -o overlap -lmpi
> totalview mpirun -a -np 4 ./overlap 192 10 &
```

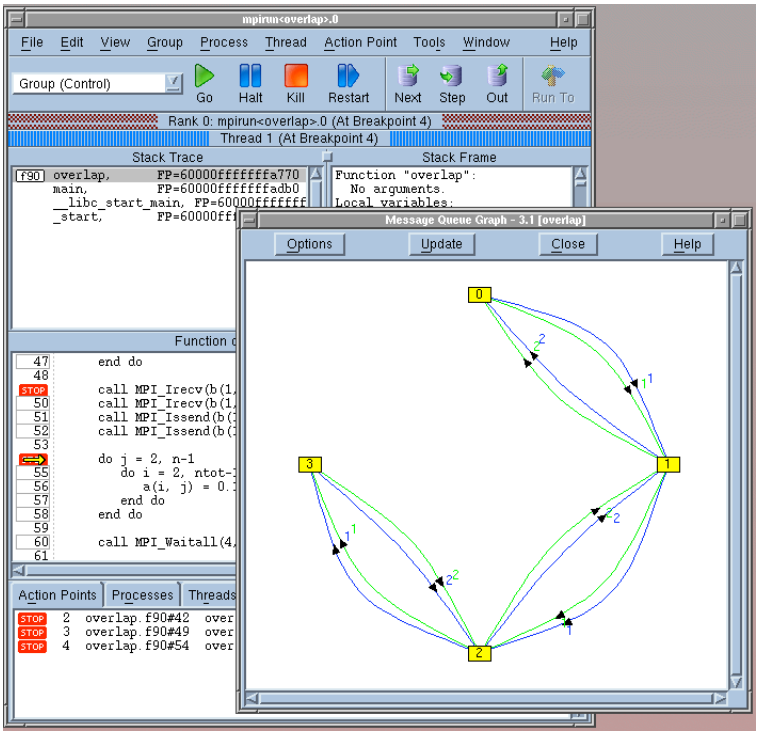
Totalview demo 1



Totalview demo 2



Totalview demo 3



Profiling

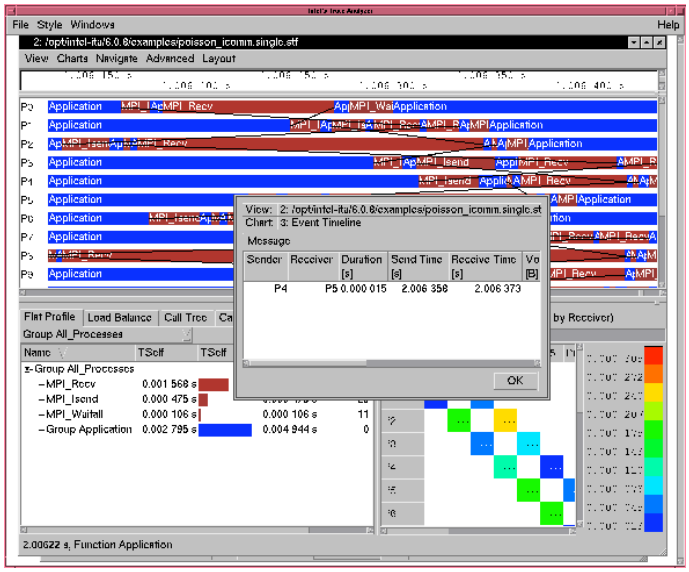
- Concerned with performance and tuning
- MPI defines a profiling interface (basically a wrapper for all MPI routines) to log message passing details. Just have to link to wrapper libraries.
- Public domain MPI profilers - upshot, nupshot, jumpshot, MPE, ...

Vampir

- Note: Vampir now owned by Intel and rename "trace collector" and "trace analyzer" - we continue to refer to tool as Vampir!
- Vampir is a parallel code profiler.
- Shows a time line of the execution of each process in your job and the communication between them
- most importantly, its shows how long each process spends in communication (MPI) procedures.
- Also able to produce statistics about communication etc.
- Simple usage:

```
> module load intel-itc
> module load intel-ita
> ifort -g overlap.f90 -o overlap -L$VT_ROOT/lib -lVT
            -lmpi  -ldwarf -lelf -lvtunwind –lnsl
            -lm –lpthread
> mpirun -np 4 ./overlap 2000 50
> traceanalyzer overlap.stf
```

Vampir screenshot



Vampir Exercise 1

Take a look at `overlap.f90` and `nooverlap.f90` and see if overlapping communication with computation provides any speed advantages. Also, which is easier to write code for? Try using Vampir to see the communication pattern. Try adjusting the amount of computation performed to see how it affects the communication pattern.

Compile the code for use with vampir

```
> module load intel-itc
> ifort overlap.f90 -o overlap -L$VT_ROOT/lib -lVT -lmpi
                    -ldwarf -lelf -lvtunwind -lnsl –lm
                    -lpthread
```

OR

```
> vampir_f90 overlap.f90  overlap
```

Run the code on a batch node

```
> qsub -q normal -lwalltime=5:00,vmem=2000mb,ncpus=4 -wd
module load intel-itc
mpirun ./overlap 4000 50
ctrl-d
```

You should have a lot of files named overlap.stf.* in your directory.

Vampir Exercise 2

Load the trace file into Vampir

```
   > module load intel-ita   > traceanalyzer overlap.stf
&
```

1. Note the relative cost of communication from the Summary chart.
2. Select "Charts/Event timeline" to see individual process activity.
3. Observe the trace and verify that computation is being overlapped with communication (zoom in) (is it really?)
4. Observe the communication pattern, is it as expected? What transfer rate are you seeing for messages? (click on messages and procedure calls)
5. Ungroup "Group MPI" to see specific MPI calls.
6. Rerun the code with `mpirun ./overlap 100 50` - is the Vampir trace expected?
7. Try all the above, except, run the code on 8 CPU's

Applications

Master-slave

- controller-worker, task-farming, ...
- Have a master process which hands out work to slave processes.
- Trivially parallel - a large set of small independent tasks rather than one large single task
- Usually don't need a high performance network - can run on cheap ethernet-based Linux cluster
- Load balance issues:
    - Don’t overload the master
    - Increase granularity by handing out batches of tasks
    - Avoid handing out the most expensive task last

Distribution of work

Master slave processes generally involve distribution of **work** rather than data. Consequently, they could be implemented without MPI with a larger number of jobs in the queue.

Exercise: Master slave

1. Using the master-slave code `masterslave.f90` adjust the granularity of work load and see how the code scales. Use Vampir to see that the communication is negligible.
2. Now compile an individual slave

   ```
   ifort -fast slave.f90 -o slave
   ```

   and run the shell script `master.sh` which solves the same problem, but submits each slave individually to the queue.

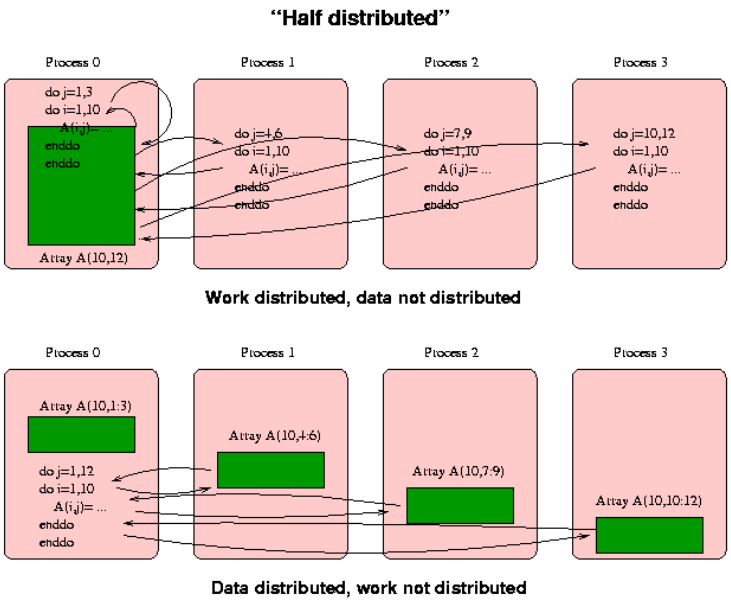
Data distribution

How do we parallelise this?

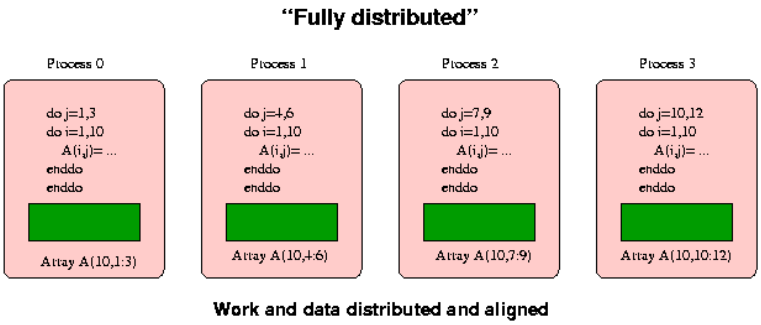
```
real A(12,10)

do j=1,12
   do i=1,10
      A(i,j) = ...
   enddo
enddo
```

Distribution of data and work



Distribution of data and work

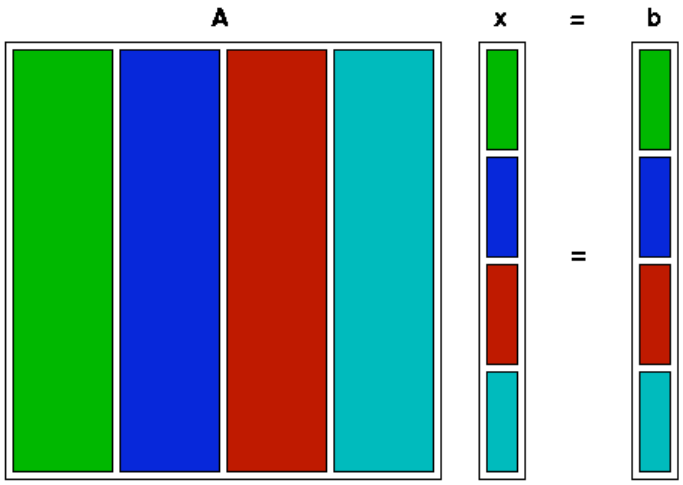


Distribution of data

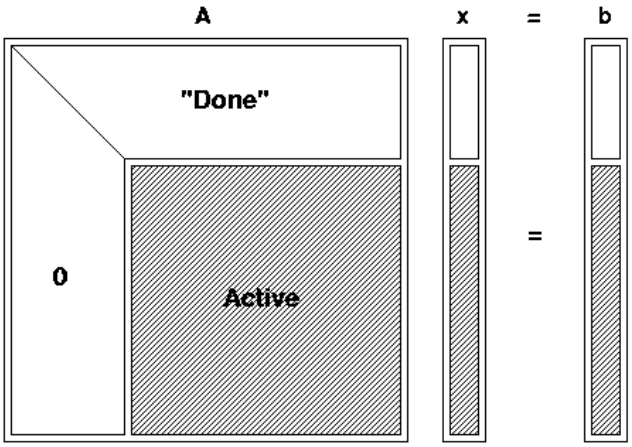
- The "all or nothing" part of MPI ...
- Want to distribute both data and work to minimize communication
- Also need to maintain load-balance at the same time.

Dense Linear Algebra

- Parallel solution of dense linear problems - LU factorisations, eigensolvers etc.
- Obvious matrix decomposition for matrix-vector multiply:

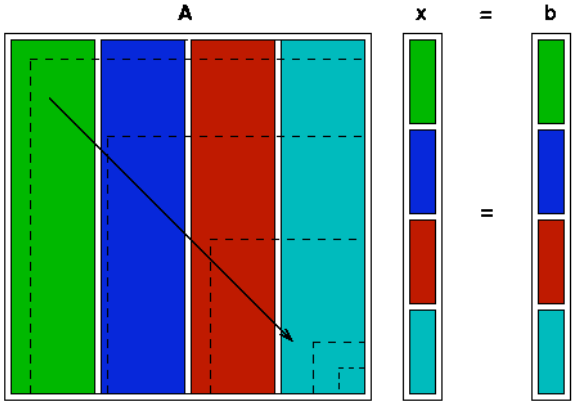


Reduction Step



Block decomposition

- Locality of reference is good - shame about the load-balance ...



Exercise 2

Compile the code

```
> vampir_f90 householder.block.f  block
> vampir_f90 householder.cyclic.f cyclic
```

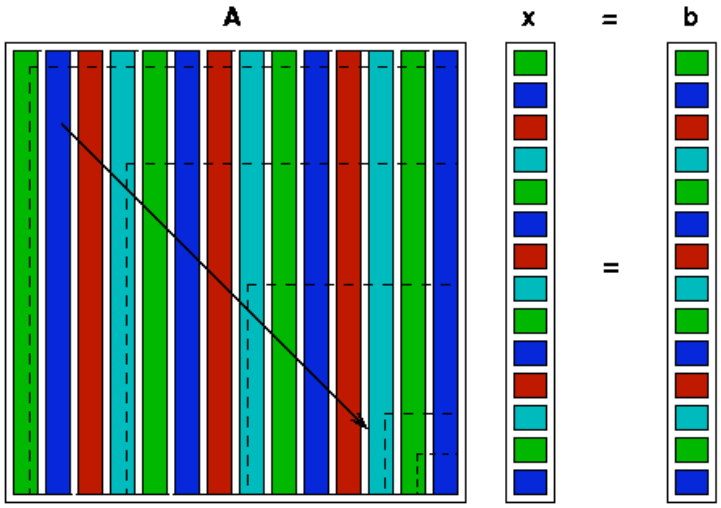
Run the code

```
> qsub -q normal -lwalltime=10:00,ncpus=4,vmem=2000mb -wd
module load intel-itc
mpirun ./block
mpirun ./cyclic
ctrl-d
```

Compare the communication patterns in Vampir

1. Compare the running time of the two different methods (look at the .o#### files)
2. Zoom in around 1s and then around 10s (drag with left button)
3. Observe the load imbalance between the block and cyclic methods (look at the Activity Chart)

Cyclic Distribution

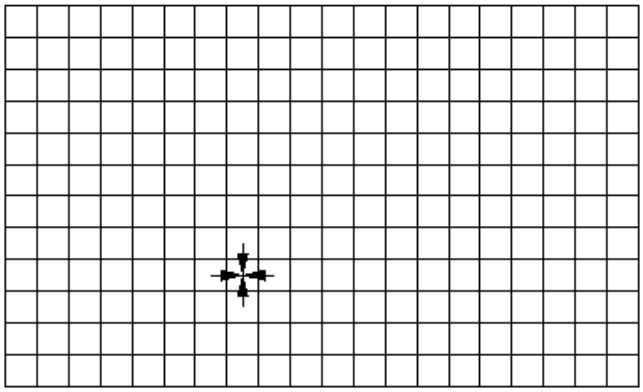


Top500

- Cache reuse very significant for BLAS2/3
- "blocked" algorithms based on fast single cpu 4x4, 8x8, ..., 64x64 matrix-matrix or
  matrix-vector operation => **block-cyclic distribution**
- ScaLapack is MPI-enabled version of LAPACK
- As good as it gets ... **8.97Tflops** on 1536 cpus of AC with problem size 170000!
- Debuted at #26 on the top500, now #200 and slipping fast ...

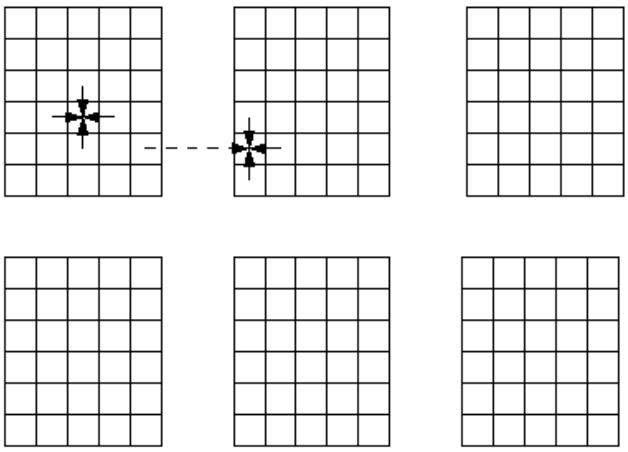
Grid methods

- See chapter 4 of Decomposing the Potentially Parallel
- Near neighbour "stencil" operations on regular cartesian mesh
- Finite difference, finite volume, ...
- Cell-centred, face-centred, vertex-centred, ...
- `v(i,j) = (1/4)*( u(i+1,j) + u(i-1,j) + u(i,j+1) + u(i,j-1) )`



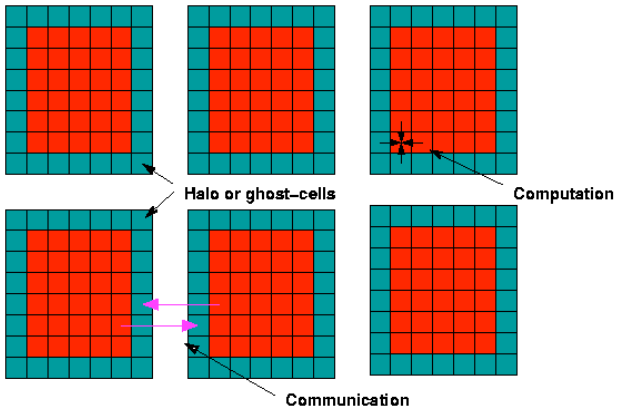
Grid methods II

- Identical work per grid point (except for boundary)
- Decompose domain evenly to processes => load-balance
- Now some stencils require off-processor data



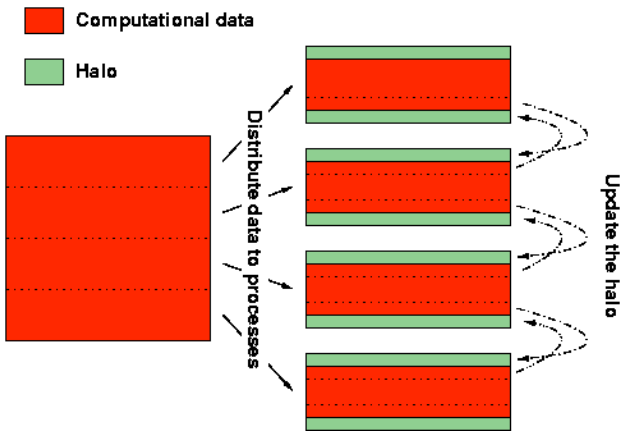
Grid methods III

- Add "halo" or "ghost-cells" to each processes domain to hold "boundary" from neighbouring processs
- Processes swap halo data before performing computation (at least around the edge)



Grid methods IV

One-dimensional "slab" decomposition:



Multidimensional decompositions

- Best theoretical communication-to-computation ratio from maximizing volume to surface area ratio of local domain
- Usually implies distributing all dimensions
- Practical issues of more small messages and strided memory access for some dimensions make analysis harder

Grid method issues

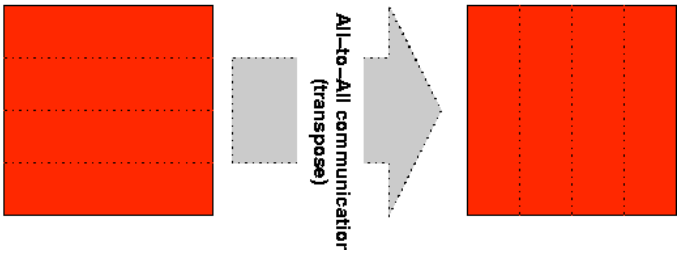
- Probably only worth going parallel for 3D problems
- Communication mode becomes irrelevant if the computation cost dominates.
- The stencil operation is usually only the "kernel" of some algorithm for solving, eg elliptic ("potential") or parabolic ("diffusion") problems. Use a good high level algorithm, eg. pre-conditioned conjugate gradient or multigrid.

Exercise: Grid methods

Missing :^)

Fourier transforms

Large communication to give locality for later calculations. Communication scales O(n) while computation scales O(n log(n)) so a slight win. Also, extra computation between FFT and FFT-1 .



Transposes

All communication is implementing transposes. Avoid extra transpose which may require you to work with transposed arrays.

See this discussion of matrix transposition.

Exercise: Fourier transforms

Time the test code `fft.f90` to see how it scales as you increase both the problem size and the number of CPU's. Use Vampir to see its communication pattern.

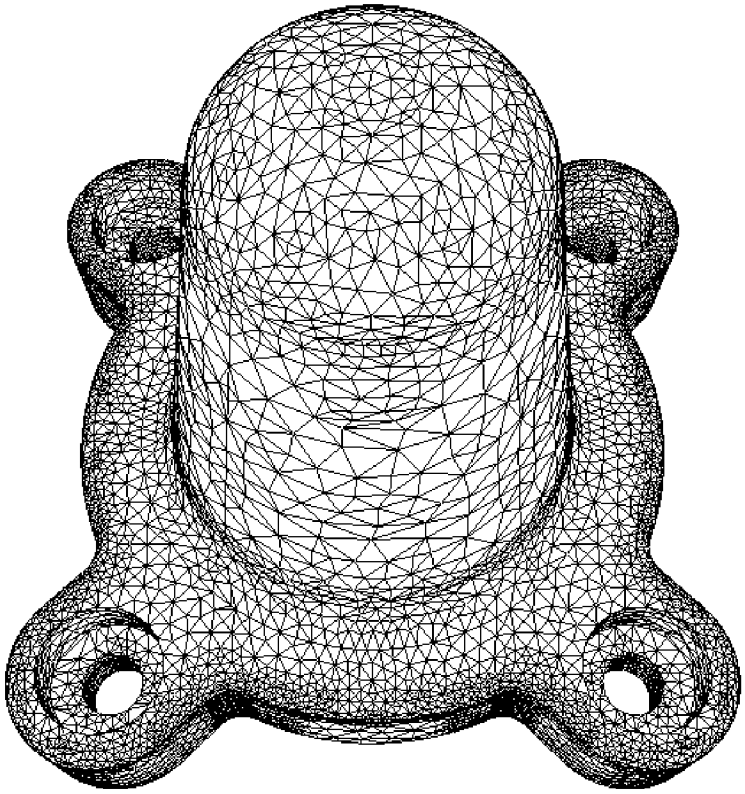
```
> ifort fft.f90 -o fft -L/opt/fftw-2.1.5/intel-9.1nf/lib
      -L$VT_ROOT/lib -lfftw_mpi -lfftw -lVT -lmpi –ldwarf
      -lelf -lvtunwind -lnsl -lm –lpthread

> qsub -lncpus=4,vmem=2gb,walltime=10:00 -wd -q express
module load intel-itc
mpirun -np 4 ./fft 1000
^D

> traceanalyzer  fft.stf
```

Unstructured Problems

How do we parallelize problems on this data?



Mesh Partitioning Methods

- Abstract maths - graph structure with nodes and links
- Communication cost with each link
- Optimization methods to minimize communication costs and still load balance computation
- Methods like *Recursive Spectral Bisection*

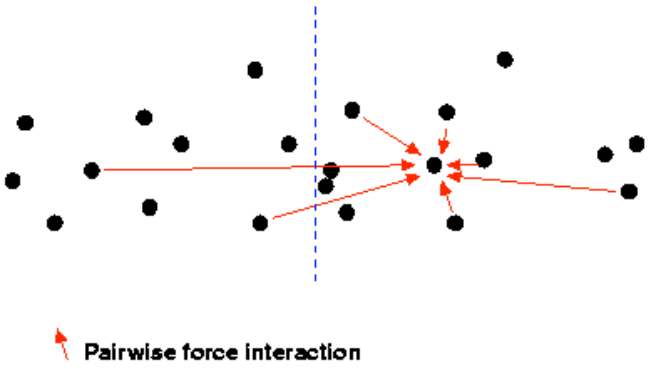
More details available in:

- Course slides from EPCC
- The later course slides from the Stuttgart HPC Centre

Partitioned Unstructured Mesh

Dynamic Unstructured Problems - Particle Methods

- Pairwise force interactions between particles
- Astrophysics N-body (all pairs interact)
- Molecular dynamics (neighbours interact)
- Data is particle position and velocity
- Work (and communication) is from interactions

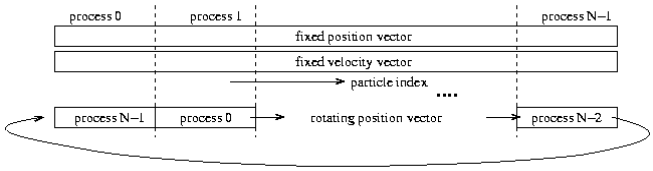


Particle Methods - Parallel issues

- Want *locality* (interacting particles local to node)
- Particles move so distribution is dynamic
- The number of interactions varies from particle to particle and changes over time
- Evenly distributing the particles does not give load balance.

Ring code

- Brute force approach for long range interaction (gravity)
- Ignores locality
- O(N^2) interactions excessive and limiting



Smart algorithm

Usually interactions only involve local particles or are weak at long range.

Smart algorithms utilise this fact to speed both computation and reduce communication.

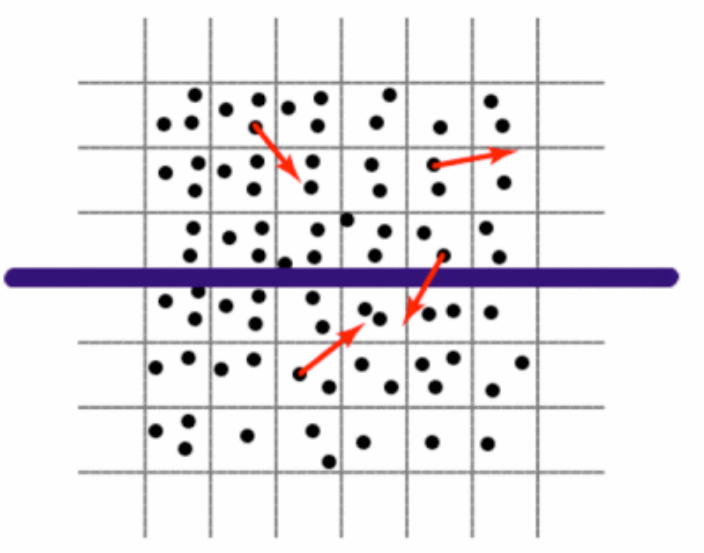
- PIC
- PPPM
- Tree codes (Barnes-Hut, Greengard-Roklin, Salmon-Warren, ...)

Much complexity from dynamic data structures for adaptive decomposition of data.

PIC - Particle In Cell

- Lennard-Jones Potential
  - short range interaction
- A cell is slightly larger than the interaction length
- Only need to interact particles in adjacent cells
- A particle can only move to an adjacent cell

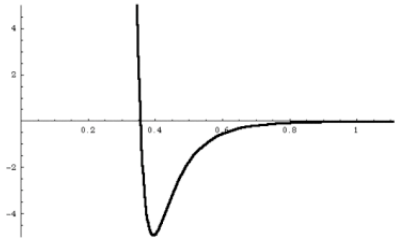
Cells



Lennard-Jones Potential

$$24\epsilon\left(\frac{2\sigma^{12}}{x^{13}} - \frac{\sigma^{6}}{x^{7}}\right)$$

- σ=0.3166
- ε=0.6502



Exercise: Molecular Dynamics

There is a parallel "pseudo-MD" code in `exercises/md/`. Only interacts with local particles and uses a "slab decomposition" to distribute cells and particles over processes.

- You will need
  `module load pgplot`
  to pick up the pgplot graphics library.
- Compile by typing `make` in the MD exercise directory
- Run the program interactively with `mpirun -np 4 ./bin/md`
- Or run in batch with `qsub test.pbs`.
- Examine the trace using `traceanalyzer`.
- `Edit the end of src/md.f90 to improve the load balancing.`
- `You can edit the Makefile to prevent plotting or to read in the particle file.`

Summary

- Get your IO right!
- Use intelligent algorithms
- Think about parallelising everything (scalabilty)
- Load balance the distributed work
- Run problems which are large compared to number of cpus, i.e. have negligible communication costs