MPI-IO Tutorial EuroPVM/MPI 2006
Part 2: Hands-on Session

Joachim Worringen

Dolphin Interconnect Solutions

<joachim@dolphinics.no>

Rob Ross

Argonne National Laboratory

<rross@mcs.anl.gov>

Motivation

- It's not understood until you've done it yourself:
  - Overcome the barrier
  - Experience the problems
  - Create a solution
  - Document what you've done

- We try to deal with real-world (*your* world?) problems.

Content

Challenge A:

- Optimize an input routine for a 3D-solver with 2D-domain-decomposition:
  - Move from sequential POSIX to parallel MPI-IO
  - Use MPI-IO with different levels of abstraction

Challenge B:

- Evaluate different techniques to create NetCDF files from within a climate code:
  - Sequential NetCDF through single I/O process
  - Parallelized through multiple NetCDF files
  - Parallelized through single file with pnetcdf/MPI-IO

  Consider the impact of the data set size!

- For both tasks, document and analyse the changing (increasing?!) performance (with support of *perfbase*).

Local System: Ubuntu Linux

- Insert DVD
- Install & start VMware Player
  - Windows & Linux Installers in directory „VMware Player“
- Copy directory „VM/pvmmpi_tutorial“ to your harddisk (3GB)
- Open contained VM with VMware player
  - Contains Linux system with a ll required tools
  - Internet access via host machine (your notebook):
    - Set up connectivity there!

- Login: '`pvmmpi06`', password '`epm2006.`'
- When opening „Terminal“ (shell) first:
  - Enter group code (4 digits, provided on „ticket“)
  - Enter IP address of perfbase server: 83.143.1.51
- Test connectivity: `ping jump.fz-juelich.de`
- For root actions: `sudo [-i]` using your own password (see above)
  - i.e. restart network: `sudo /etc/init.d/networking restart`

Remote HPC System: JUMP

- IBM p690 cluster „JUMP“ at Forschungszentrum Jülich
  - URL: `http://jumpdoc.fz-juelich.de/`
  - For us: single SMP node with 32 CPUs (shared with MPI-2 tutorial)
  - GPFS file system
- **Access** via ssh: `ssh -X mpiXXXX@jump.fz-juelich.de`
  - `XXXX` is your four-digit user code
- Within Virtual Machine:
  - no password needed
  - More convenient: use `jsh` (no arguments)
  - Remote copy with scp wrapper:

```
jput file [files...] :remote/path
jget remotefile [files...] :local/path
```

    - Target paths are relative to $HOME

Usage Hints

- Edit files locally: editor emacs
- Transfer source files via **`jput`** , compile & run, get results via **`jget`**
- Compilers for MPI code:

	Local	Remote
C:	**`mpicc`**	**`mpcc`**
Fortran 90:	-	**`mpxlf90`**

- Interactive Execution:

  Local: **`<executable> -np # <arguments>`**

  Remote: **`llrun -p # <executable> <arguments>`**

perfbase - Overview

- Tracking performance development made easy
  - Define *experiments* with parameter and result values
  - Feed data into experiments from arbitrary output
  - Analyse and visualize data with arbitrary queries
  - GPL Open Source: http://perfbase.tigris.org
- Usage here:
  - Experiments are already defined and setup in a common database:
    - íoperf
    - wrarray
  - Capture benchmark output at JUMP (use `runjobs` script)
  - Transfer it to local machine (`jget`) and feed into the experiment
  - Perform queries using predefined query descriptions

Install/Update perfbase

- Native Linux:
  - Get http://www.pvmmpi06/perfbase.tar.gz
  - Unpack, run setup
    - Install missing Python modules if necessary (using yum, apt-get, ...)
- Update in VM:
  - Activate folder sharing
  - Copy over perfbase.tar.gz
  - Unpack as root to /opt:

```
cd /opt; sudo tar xzf /path/to/perfbase.tar.gz
```

perfbase Setup & Usage

- Address of perfbase server might need to be updated:

  ```
  echo <ip-address> >~/.dbhost
  ```

- Test if perfbase is reachable by listing all available experiments:

  ```
  perfbase info -a
  ```

  – Experiments for this Tutorial are 'ioperf' and 'wrarray'

- More information on an experiment (i.e. *ioperf*):

  ```
  perfbase info -e ioperf
  ```

- Import data for experiment „**exp**“ (*ioperf* or *rdarray)*:

  ```
  perfbase input -e <exp> -n <exp> datafile [datafiles...]
  ```

  i.e.:

  ```
  perfbase input -e ioperf -n ioperf parallel_np8.dat
  ```

- Queries:

  – i.e. „show maximum bandwidth in ioperf across all modes and process counts for default data set size for runs on Jump“:

  ```
  perfbase query -e ioperf -n max_bw
  ```

Hello world!

Transfer code and login:

```
jcp -r hello_world :
jsh
```

C:

```
cd hello_world
mpcc hello_world.c -o hello_worldc
llrun -p 2 hello_worldc
FZJ-INF: @data limit=3.000GB, @stack limit=0.500GB inserted
llsubmit: Processed command file through Submit Filter: "/home1/admin/loadl/filter".
ATTENTION: 0031-408  2 tasks allocated by LoadLeveler, continuing...
Hello world! I'm 0 of 2 on j39
Hello world! I'm 1 of 2 on j39
```

Also for Fortran:

```
mpxlf90 hello_world.f90 -o hello_worldf
```

Oops – does not compile! Fix is needed – edit the file. Look at the bottom of the file.

```
llrun -p 2 hello_worldf
```

Maybe another oops here: crash! Did you take a close look at the comment?

ioperf

- ioperf is derived from the I/O part of a real-world climate simulation
  - Written by Rene Redler of C&C Research Labs, NEC Europe Ltd. - thanks!
  - In climate simulation, NetCDF is a very popular data format
  - Task of ioperf: store 3D array distributed in 2D in file(s)

- Until recently, only serial versions of the NetCDF library existed
  - pnetcdf has changed this by using MPI-IO
- Typical I/O (write) strategies with serial NetCDF for parallel applications:
  - „master I/O“: all processes send data to process 0, which writes it to a single file
    - Problems: strictly sequential I/O; memory requirements
  - „manyfile I/O“: each process writes it's own data file
    - Problems: management of the files; fixed number of processes
- Natural strategy for parallel NetCDF:
  - All processes write data in parallel into a single file
    - Problems: ?

Ioperf implements all these strategies and allows comparison!

ioperf – Task overview

- Compile & run ioperf on JUMP only (not locally)

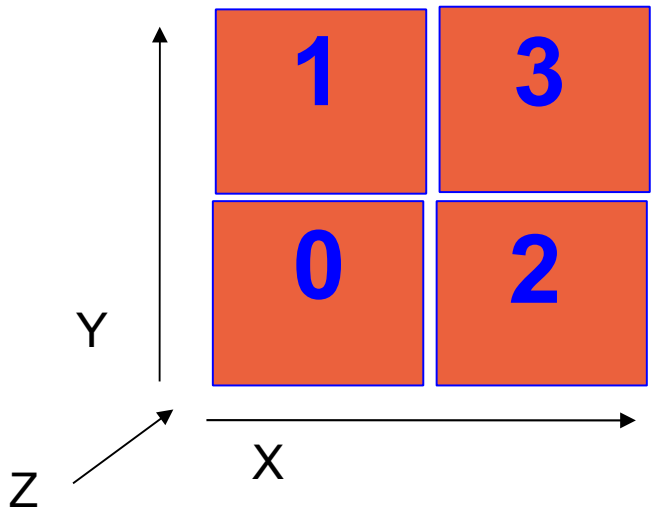
- **Step 1:** Compile (`make`) and see if any problems show up. If yes, fix them by editing the file at the listed function call.
- **Step 2:** Test execution (`llrun -p 1 ioperf`) and see if any problems show up. Fix them by looking close at the same location as in step 1.
- **Step 3:** Run the serial NetCDF version:
  - Edit namelist.ioperf: remove '!' in front of „serial“ (uncomment it, comment out “multi” and “parallel”)
  - runjobs <np> <executable>
- **Step 4:** Run the pseudo-parallel NetCDF version:
  - Edit namelist.ioperf: remove '!' in front of „multi“ (uncomment it, comment out “serial” and “parallel”)
  - runjobs <np> <executable>

ioperf – Task overview (cont'd)

- **Step 5:** Run the really parallel pnetCDF version:
  - Edit ioperf.F90 and fill in the missing code (pnetCDF calls) in subroutine „richman“ (search for „remove“)
    - Look at the related calls in subroutine „poorman“
  - Edit namelist.ioperf: remove '!' in front of „parallel“ (uncomment it, comment out “serial” and “multi”)
  - `runjobs <np> <executable>`
- **Step 6:** Transfer the result files to the local machine and import it into perfbase:
  - `perfbase input -e ioperf -n ioperf`
- **Step 7:** Perform some queries:
  - `perfbase input -e ioperf -n max_all`
  - `perfbase info -e ioperf -v` shows other available queries
- **Optional steps:**
  - Change the grid resolution in ioperf.F90:
    - Decrease value of `dy_lat` from 8.0 to some smaller value
  - Import data into perfbase and re-execute queries with matching dimensions:
    - `perfbase input -e ioperf -n max_all -f f:i=I,f:j=J,f:k=K`

wrarray

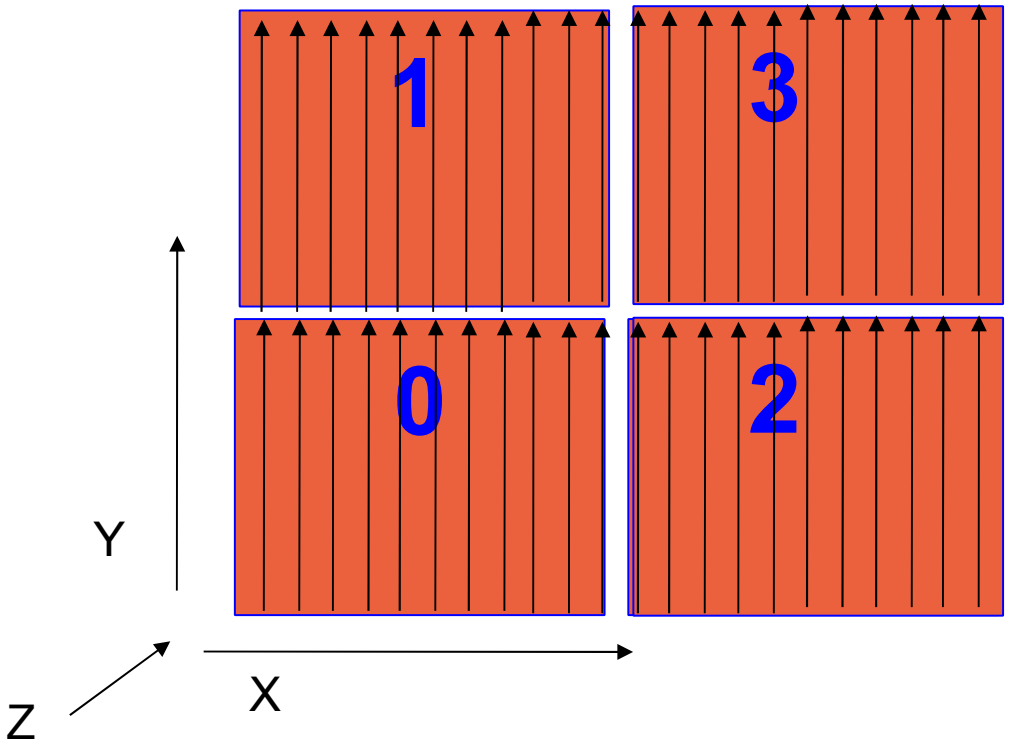
- A 3D-array is distributed across processes:
  - 2D-decomposition along X- and Y-dimension
  - Z-dimension remains completely on each process



- Data order in Array is C: A[x][y][z]
- Order in memory (of one process):
  x0,y0,z0 x0,y0,z1 x0,y1,z0 x0,y1,z1 x1,y0,z0 x1,y0,z1 x1,y1,z0 x1,y1,z1
- Order in file should be globally increasing!

Storing to File

- Many storage operations required!
  - Loop over X dimension
  - In each iteration, `seek()`, then `write()`
- Or can we avoid this?



wrarray Task Overview

- **Step 1:** Create POSIX variant and verification tool:

  `make posix verify`

  – Will create '`wrarray_posix`' and '`wrarray_verify`'
- **Step 2:** Test POSIX variant with default settings:
  – Use a small number of processes and no arguments:

    `llrun -p 1 wrarray_posix`
  – Run verification tool: `./wrarray_verify`
- **Step 3:** Vary parameters:
  – Set dimension extents via -x <n>, -y <n>, -z <n>; iterations via -i <n>; filename via -f <name>
    - File size: x * y * z * 8 bytes * iterations
  – Create very small file and check content manually:

    ```
    wrarray_posix -x 4 -y 4 -z 3 -f tiny.dat
    od -t f8 tiny.dat
    ```
  – Definition of 3 standard file sizes to be used:

Small:	-x 128 -y 64 -z 32	2MB filesize
Default:	-x 512 -y 256 -z 128	128MB filesize (built-in defaults)
Large:	-x 1024 -y 1024 -z 128	1GB filesize

(„Large“ is not large for this machine, but for our time budget)

wrarray Task Overview (cont'd)

- **Step 4:** run a series of jobs with POSIX (distribute node counts among participants, if possible!):

  `runjobs <NP> wrarray_posix`

  – Output will be stored in *.out files
- **Step 5a:** Port the POSIX I/O routine to simplest possible MPI-IO:
  – Edit wrarray_simple-mpiio.c
  – Port over the POSIX functions to their MPI_File_... equivalents
  – Run & test your code with `wrarray_verify`
  – Run a series of jobs like above:

  `runjobs <NP> wrarray_simple-mpiio`
- **Step 5b**: Port the POSIX I/O routine to collective MPI-IO:
  – Edit `wrarray_coll-mpiio.c`
  – Port over the POSIX functions to their **collective** MPI_File_... equivalents
  – Run & test your code with `wrarray_verify`
  – Run a series of jobs like above:

  `runjobs <NP> wrarray_coll-mpiio`

wrarray Task Overview (cont'd)

- **Step 5c:** Port the POSIX I/O routine to optimal MPI-IO, using its full potential:
  - Edit `wrarray_full-mpiio.c`
  - Guideline:
    - We need to get rid of the loop over the I/O calls for better performance!
    - Tell MPI-IO more about the global data structure that will be written:
      - Construct an MPI datatype that describes the global array, and the local part of it.
      - Set up a file view with this datatype: Every process will only „see“ the part of the file that it needs to access.
      - Issue a single collective write operation, submitting all your local data to the MPI library. The MPI library will care for the rest!
  - Run & test your code with `wrarray_verify`
  - Run a series of jobs like above:

```
runjobs <NP> wrarray_full-mpiio
```

wrarray Task Overview (cont'd)

- **Step 6:** Move the data into your virtual machine and import it into perfbase:
  - Alternatively, send a tar file to joachim@dolphinics.com
  - Import data into perfbase:

    ```
    perfbase input -e wrarray -n wrarray <your file>
    ```
  - Perform queries:

    ```
    perfbase query -e wrarray -n max_all
    ```
    - Other queries: min_all, stddev_all
- **Optional steps:**
  - Vary the data size to „large“ (or „small“) and rerun code
  - Import the data into perfbase and compare with other results:
    - Set x, y, z to the values you want to see (replace '<N>' with number):

    ```
    perfbase query -e wrarray -n min_all -f f:x=<X>,f:y=<Y>,f:z=<Z>
    ```