Title : ***Cluster Tools***

Authors : Colin Enticott AND Julian Crooke

School of Computer Science and Software Engineering
Monash University
Caulfield Campus, Melbourne, Australia

Email : cme@csse.monash.edu.au
jcrooke@cs.monash.edu.au

0. Abstract

Cluster tools are tools used on clusters of computers. The nature of these tools ranges from programming languages and variations on current languages (libraries or compilers) to tools assisting in design, optimisation, and debugging of parallel programs. There are synchronisation problems that can occur in a program that do not happen in each of the execution times, only the important ones. Tools come in many flavours, such as: diagnostic or debugging, performance-tuning and execution-tools. All of these tools are designed to assist the programmer achieve their goal quickly and accurately.

This report will discuss briefly general issues with cluster tools as well as discussing three of them in detail. The first is a debugging tool called “Message Queue Monitoring” [1] that displays the message queues on the entire cluster computer system graphically. The second we’ll discuss a set of “Parallel Unix commands” [2] tools for maintain the files and Unix commands on a cluster of Unix machines. The third is a tool that coordinates the output off all the nodes on the cluster to one host in an organise fashion, “Parallel print functions” [3].

A few other noteworthy cluster tools are mentioned near the end.

1. Introduction

Parallel programming over a cluster of computers enables high performance computing on a network of computers. Distributed programming has been around for many years with client/server implementations of past software, but these implementations have a very limited in scalability as the one server does most of the processing. One of the goals of cluster computing is still the same as the client/server approach, that is to have the end user think that they are just using the computer in front of them only, and not many machines working together. This can even be brought down to the programmer level, the programmer writes one program and unknowingly, sections of the program execute on different machines on the network, but all the results and output appears on the terminal the programmer executed the program on. This can be done with Joyce-Linda if the programmer is unaware of the implementation, but is fully aware on how to use the "agent" command.

Cluster tools help in the designing and implementation of parallel programs including cluster computers. The tools range from programming languages (Joyce-Linda, etc.), to variations on existing languages (libraries or enhanced compilers), and to tools assisting in design and debugging of parallel programs (parallel debuggers (e.g. pdbx, pedb), parallel tracers (e.g. VAMPIR) and profilers (e.g. xprofiler) [all from 12]).

There are different types of tools used in cluster programming, the most common of them being diagnostic or debugging tools. These tools assist programmers in analysing problems that cluster and parallel programs have. The most common cluster programming difficulty must be deadlocks. One tool that is geared toward the isolation and eradication of deadlocks, mentioned further down, is "Message Queue Monitoring". This graphically displays the state of the whole cluster application message state.

Another area in cluster computing tools is performance-tuning tools. One such tool might be a timing API, or a profiler. Cluster computing can involve any of several different node architectures. One may wish to exploit this by having tasks executing on machine architectures suited to that type of processing. But which routines run best on which architectures? Accurate cross platform timing routines with a standard interface would be the best method of finding what runs best where (guessing the performance of an architecture given a particular problem or exact program implementation is not often going to be accurate enough, only a rough guide). Other such performance-tuning tools may include message passing optimisation and load distribution tools.

Another group of tools are execution tools. These tools help in the execution of programs across the cluster of computers. These tools may include a program that shows the processes state across many computers, or a given computer, copy files to other cluster computers and possibly begin compilation on the cluster computers (the Parallel Unix Commands could be considered execution tools).

2. Works on this area

Name	Description	Remarks	URL
Message Queue Manager project	Parallel tool that displays a parallel programs message queue	Debugging and performance tuning	http://www.ptools.org/projects/mqm/
Parallel Unix commands	Parallel versions of familiar Unix commands	For execution and setup of parallel programs	http://www-unix.mcs.anl.gov/sut/
Parallel print functions	Standard API for merging and identifying the output from multiple nodes in a parallel program	Parallel program output	http://www.llnl.gov/sccd/lc/ppf/
Portable Timing Routines	A standard API (application program interface) for measuring intervals of program execution, in terms of wallclock, user CPU, and system CPU time	Standards for multi-types of processors performance timing.	http://www.ptools.org/projects/ptr/
Distributed Array Query and Visualization	Tools providing access to, and visualization of, distributed arrays in a parallel program	Used for debugging	http://www.cs.uoregon.edu/~hacks/research/daqv/
Performance Data Standard and API	A standard API (application program interface) for obtaining the values of hardware counters	Performance tuning	http://icl.cs.utk.edu/projects/papi/
Dynamic Probe Class Library	A standard infrastructure that will make it possible to build tools that are portable across HPC platforms	Tool building	http://www.ptools.org/projects/dpcl/

3. Message Queue Manager project

Problem

One of the common problems programmers have with distributed programming is a deadlock. A deadlock occurs if each process running a distributed program is waiting for information to be sent from another process. Given that they are all waiting, no task sends any information to satisfy another, and the program waits indefinitely. There is no quick fix for these deadlocking problems, and they can be dealt with in differing ways.

A possible solution

One way is to have a controlling node that instructs each of the nodes what to do and receives the results from a node when it is done. This has the added advantage of the programmer thinking of the whole application linearly. It is rare for deadlocks to occur when the each node does a task almost synchronously. This controlling node can be used for debugging as is can output the messages it received which lets the programmer use the information to find when it did not receive an important message. The only problem with this is a whole node is wasted to just pass messages and messages are usually doubled up as the controller node passes the results from a node to another node. A node will run far more efficiently if it knows what needs to be processed next and the communications network between the nodes is quicker because it is used far less.

The project

The “Message Queue Manager project” created a program that is designed to graphically display the messages to and from each node so that a programmer can easily check when and where deadlocks occur. The output from this program is like the output that could be obtained from a controlling node, but does not have the network overhead. And it is also displayed graphically so the programmer does not have to wade though mounds and mounds of output.

The program is designed for ease of use. The main window (figure 1) consists of a grid that represents all the nodes of the application. Each square in the grid has a colour that represents the number of messages in the process queue of each node. A node can be selected and a new window that shows that nodes operation. The new windows contains two mode grids, a ‘from’ and a ‘to’ grid, and filters to display certain messages. This second window can represent a node or a selection of nodes.

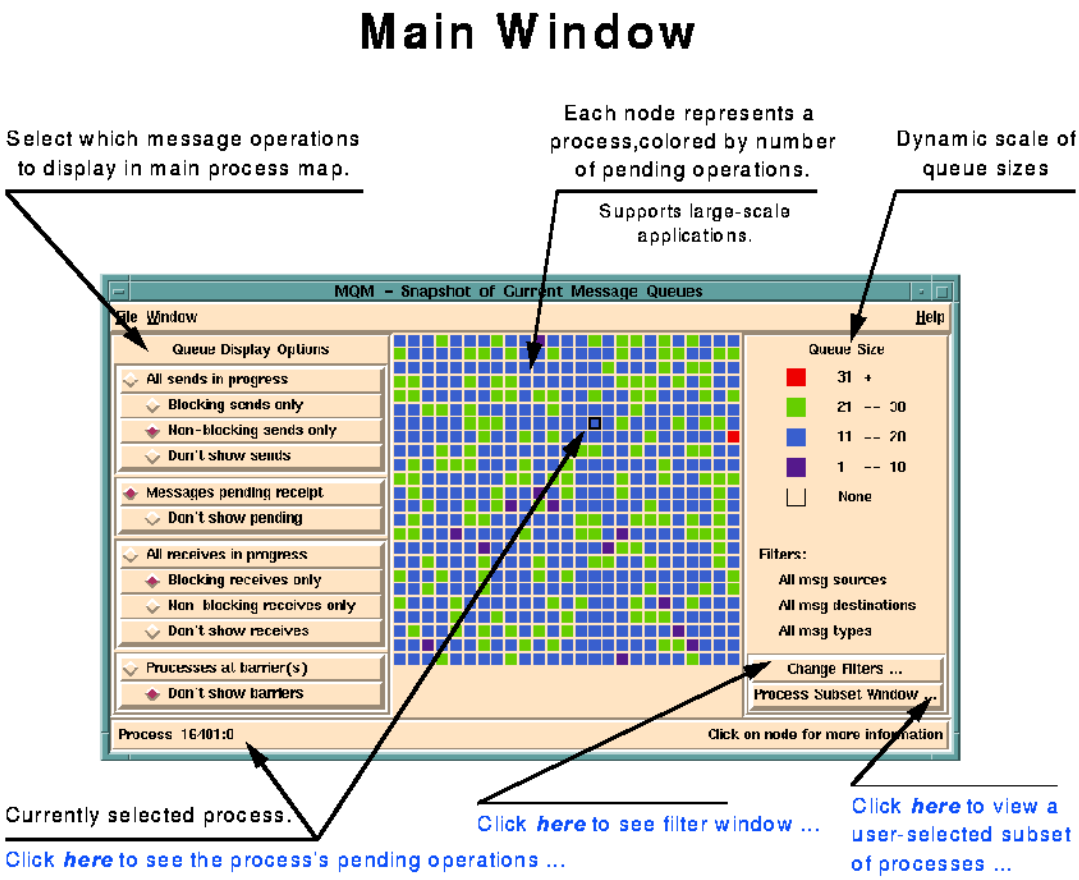


Figure 1 - Main window of "Message Queue Manager" [4]

Being able to analyse each node or groups of nodes makes it easier for the programmer to isolate the problem the application is having with distributed design. One such problem would be the dining philosophers problem. If each node was waiting for it's previous node and the first node is waiting on the last node, with then manner of a few clicks, it can easily be determined leaving the programmer to implement a work around, or, re-think the problem.

In cluster computing this is a very valuable tool. It helps programmers isolate problems in their cluster program rapidly and accurately. There is no need for guesswork as programmers can check the current and final state of all the messages in the application. Guesswork normally leads to placing run-time reporting in the program that requires a recompile. Recompilation on a cluster of machines can take a while if there are different compiled files on each machine, as each machine has to be instructed to recompile. Diagnostic tools like this one will speed up the finding the cause of deadlocks.

The "Message Queue Manager project" software is available in source code for Tcl/Tk and C and it is royalty-free. An implementation has been successfully integrated with Intel's Interactive Parallel Debugger for the Paragon.

4. Parallel Unix Commands

Problem

Working in a cluster environment, most users need to perform file maintenance and process monitoring on more than one node at a time. The user may for example wish to copy some updated source files to all nodes in the cluster, then build the binary executable separately on each node. Logging in to each of the machines and uploading files and executing the commands individually would take forever. The sequence of commands may be different every time, which may require interactivity, so scripting is not a desired option. On a regular basis, this problem may grow very tiresome for the user. What is needed is a way of telling many nodes to execute the same command at once, and that is easy to learn and use.

Concept for solution

The Parallel Tools Consortium (Ptools) created a workgroup (Parallel Tools Consortium Working Group on Parallel Unix Commands) to make a solution to these problems. The solution they came up with is designed to work for a cluster of Unix machines, and it is a set of commands (shell commands) that run on several nodes at once. These include 'pls', 'pcp', 'pmv', 'pps', and more. Each command starts with 'p' for 'parallel' and then usually follows with the name of a standard Unix command ('pcp' = parallel copy). The syntax is always that the first parameter to the parallel command is a node list. This can be '-all' to specify all nodes, and is discussed in more detail later.

The semantics do not follow a completely rigid rule, because it is practical sometimes not to. For example, running 'pcp –all newData ./inputData' will copy the local file 'newData' to file './inputData' in all the nodes. Then, running 'pmv –all inputData myprog/data' will move the file on each local hard-drive of each node from the current directory to 'myprog/data'. Clarifying, the pcp 'broadcasts' or distributes the file newData, while pmv moves the file within each hard-disk, locally only. This is over all probably the most useful behaviour of these commands. One rarely needs two copies of something on each of 100 nodes after all, and it is a very quick, clear, and reliable way to spread a file across the whole cluster. On the other hand one never really wants to move a single file from one node to all other nodes (besides, that effect could be achieved with a 'pcp' followed by a local 'rm' anyway). Moving the same files on each node into another directory on each node is a much more commonly desired operation.

Syntax of the commands

As mentioned earlier, all commands start with a ‘p’, and the first argument to each command is a nodelist as described in the grammar below:

```
nodelist -> ’-all’
nodelist -> range [ , nodelist ]
nodelist -> nodenum [ , nodelist ]
range    -> nodenum ’-’ nodenum
nodenum  -> digit [ nodenum ]
```

This grammar describes the first argument common to all the parallel commands in the project [from http://www-unix.mcs.anl.gov/sut/long2/node3.html] Some examples are:

```
‘pcp 1-5 testfile /progfiles’
```

This copies testfile to /progfiles on nodes 1 through 5 only.

```
‘pmv 1,3,5 /progfiles/testfile /progfiles/old’
```

Copy testfile to /progfiles/ on nodes 1, 3 and 5 only.

```
‘pps machineA’
```

List the processes running on the node named ‘machineA’ only.

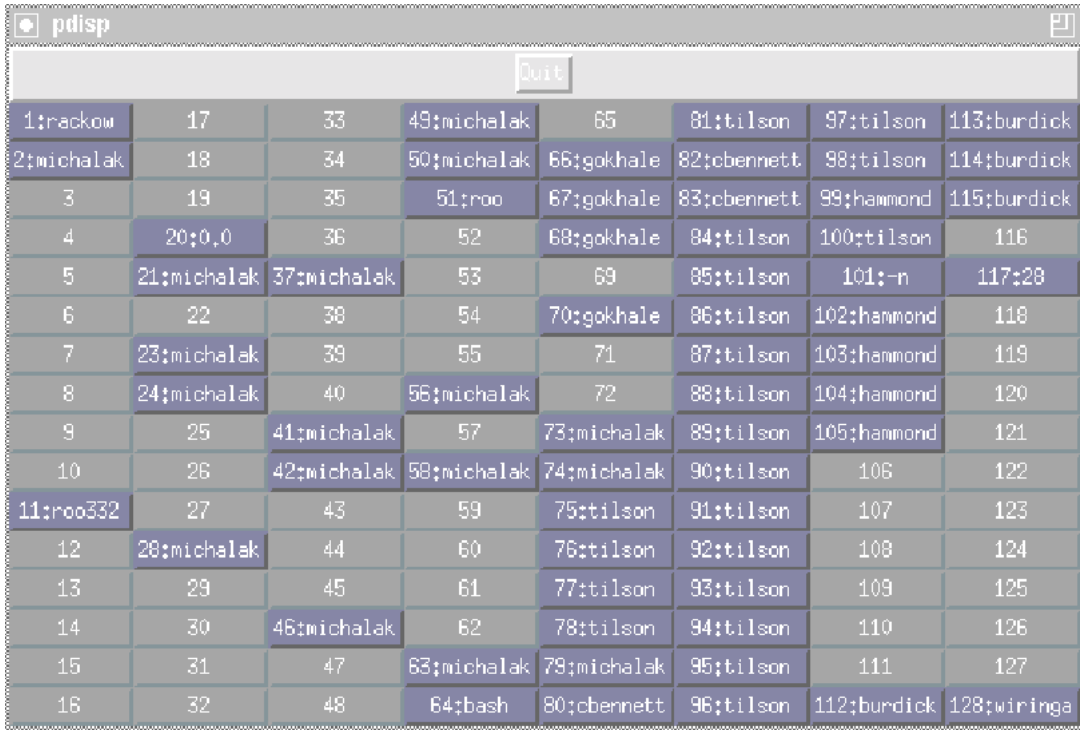
```
‘pps 5,machineA,7-10’
```

List the processes running on node 5, machineA, and nodes 7 through 10.

Graphical display components

For commands which print output, the node number or name prefixes each line of output. Some commands' outputs can be passed into graphical visualization tools, such as *'pdisp'* and *'pinfo'*:

Pdisp **used here to display nodes on which a certain job is running (text is piped in to pdisp from other parallel commands) [from 7]:**



Pinfo used here to display 'partition' availability [from 7]:

Speed of operation

In keeping with the requirement that the commands can be used interactively, they must start quickly and end quickly to return control back to the user for another command. Otherwise, as mentioned in the Ptools proposal [6], users will not use the commands as casually as they should be able to and will feel compelled not to use them as much. This is addressed in the tool design, in a scalable fashion. The command, along with half of the node list, is sent to one of the nodes. This is repeated until there are no nodes left in the list. (So initially one node holds the complete node list to distribute the command to. It passes half of that list to a second node. The second node does the same thing with its list, while the first node does the same again with its remaining list. At step, the number of nodes the command has spread to doubles. So in **O(log n)** steps, the command has been passed to all **n** nodes. They return the 'finished' message with similar haste, before the user may enter another command.)

Evaluation

Here is the list of available commands, and a brief inspection of their functionality is available also [7].

Table of commands included in the project at time of proposal [from 7]:

Program Name	Action
`pcp`	Parallel copy (for systems with local disks on each node).
`pcat`	Parallel concatenation of files
`pls`	Parallel directory list (`ls`).
`prm`	Parallel remove
`pmv`	Parallel move
`pfind`	Parallel `find`
`pps`	Parallel `ps`
`pfps`	Parallel process find
`pkill`	Parallel process kill
`pexec`	Run a command on all selected processors
`ptest`	Run `test` on all selected processors, anding the results and returning a single status value.
`ppred`	Run a command when a condition is satisfied
`pdistrib`	Run a command on a collection of files
`pdisp`	Display the output of a command graphically

That is fairly good selection of powerful commands to have at one's fingertips. As seen, graphical front ends can be added with ease (pdisp comes as part of the project) due to the simple output of the commands. For setting up and maintaining files and processes in a cluster, this is a very useful tool.

5. Parallel Print Function

Problem

Programming parallel programs can involve the use of special parallel debuggers and analysis tools. But many people prefer to work with print statements [7]. They are usually scattered through code to indicate arrival at execution checkpoints, communication events, intermediate results and so on. The actions or states of different processes are printed in this way through the execution of the program. This output can be used to analyse the flow of data and events, but often the volume of ASCII data that comes from several nodes all printing out the same messages can hamper efforts to investigate any problem at hand. One must be perpetually attentive to print-statement placement in order to avoid printing things over and over. One often needs to add extra code to every print-statement which prints the number (or name) of the originating node:

e.g.

```
printf("Node %d says hello there.\n", myRank);
```

This code running on 3 nodes would produce something like:

```
Node 1 says hello there.
Node 0 says hello there.
Node 2 says hello there.
```

The repeat printing problem can be easily solved for cases where only one particular process will print the message:

```
if ( myRank == 1 )
      printf("Node %d says hello there.\n", myRank);
```

But there are often cases where we can't know easily what is going to be printed by each node. For example, say each node could detect whether its CPU was overheating. We print the status of the CPU:

```
if ( meOverHeating )
      printf("Node %d says Water Please!\n", myRank);
else
      printf("Node %d says I'm Cool.\n", myRank);
```

This will print something like this when run on 4 nodes:

```
Node 0 says I’m Cool.
Node 1 says Water Please!
Node 3 says I’m Cool.
Node 2 says Water Please!
```

Which follows no sequence, and expresses the information inefficiently. What we really wanted was probably something more like this:

```
Nodes 0 and 3 say I’m Cool.
Nodes 1 and 2 say Water Please!
```

Reducing output by two lines.

Another common trait of print-statements in parallel applications is their unfortunate tendency to interrupt one another mid-line resulting with text inserted at strange points. This creates output that is consistently harder to read and may possibly be truly ambiguous, for example (a tragic rerun of before):

```
Node 0 says Node 1 says Water Please!
Node 2 says I’m Cool.
Water Please!
Node 3 says I’m Cool.
```

This is already quite hard to follow with only three nodes involved (albeit an unlikely pathological case). But fear not, for we will see a tool offered as a solution to all these problems, which adheres to sensible requirements of user familiarity and ease of use.

A possible solution

Ptools created a workgroup (Parallel Tools Consortium Working Group on PPF - Parallel Print Function) to satisfy the problem at hand.

The group had the realization that if all N nodes are printing “hello there”, then there is no reason to actually print “hello there”, N times over. (This is confirmed in the study of information theory, from which arises the result that a repeated message has low entropy, which implies low information content.) Since this is a major problem with conventional printing, it seems sensible that the PPF’s primary function should be to collapse identical messages from multiple nodes down to one message. The single message will contain a list of nodes (ID or rank) at the beginning, indicating which tasks printed the message.

e.g.

```
PPF_Print(MPI_COMM_WORLD, “Node says hello there.\n”);
```

When run on 3 tasks, this would output:

```
0-2 Node says hello there.
```

Note that this is doubly optimized, firstly the printed lines are collapsed from three down to one.

Secondly the list of nodes at the start of the line is briefly expressed as a range (similar to the input node lists in Parallel Unix Commands).

Note also the MPI_COMM_WORLD. This determines which nodes are printing. All of the nodes in the specified communicator must call parallel print, otherwise the program will hang, if for some nodes we don't want to print anything, NULL can be passed in as the format string, and the node has then satisfied the call, but not printed anything. Even though all the processes must call the print before moving on, this does *not* imply that the PPF synchronises like a barrier.

Here is an example where we use MPI_COMM_SELF instead:

```
PPF_Print(MPI_COMM_SELF, "Node says hello there.\n");
```

Run on 3 nodes would produce something like:

```
2 Node says hello there.
0 Node says hello there.
1 Node says hello there.
```

Any grouping and ordering is dissolved because the prints were all to separate communicators.

One more feature is the use of a **%N** format specifier to set the position to print the node list:

```
PPF_Print(MPI_COMM_WORLD, "Node %N says hello there.\n");
```

Run on 3 nodes would produce something like:

```
Node 0-2 says hello there.
```

Evaluation

Apparently the input format strings may be no longer than 255 characters, none of which may be of the value 1 or 2 (these are reserved as internal markers). Also, only one **%N** can be added to the format string. These are the limitations mentioned in the prospectus [8], but each are very mild limitations indeed.

The tool is based on MPI specifically in C (printf) and Fortran (PRINT), but is MPI implementation-independent. This means it is fairly widely available and quite portable. Having been tested and working on several systems including an IBM RS6000/SP and a DEC Alpha cluster (each with two implementations of MPI), PPF appears a worthwhile addition to any MPI programmer's toolkit.

6. Success Stories

The “Message Queue Manager project”[5] has been successfully integrated with Intel's Interactive Parallel Debugger for the Paragon.

Another tool worth mentioning perhaps is GA, “Global Arrays toolkit”[9]. It implements an efficient way of storing and accessing a distributed array as though it were stored on a shared memory architecture. Numerous awards associated with the project have been presented to the vendors.

“Condor”[10] is another noteworthy project - a tool aimed at seeking idle nodes in a network and using their spare cycles to compute some specified problem. (“Condor - a hunter of idle workstations” is the title of one conference paper regarding it [*not referenced*].)

7. Summary and Conclusions

There are three types of parallel tools that we discussed in this document. Diagnostic or debugging, performance-tuning and execution tools. Each of these tools helps programmer to achieve reliable software rapidly.

The Ptools consortium designed and created all three of the tools we introduced, to aid the debugging of cluster programming. By displaying the message queues for all nodes graphically with MQM, deadlocks and other problems can be identified and solutions derived. Parallel Unix Commands tools help set up, rationalize, and maintain the overall configuration of the nodes in a cluster. And PPF smooths the progress of tracking down errors in the ways users are accustomed to when programming in MPI.

Tools like these are likely to be valuable if the development of large distributed computing applications is to be carried out on schedule and within monetary constraints.

8. References

[1] "Message Queue Manager project" - http://www.ptools.org/projects/mqm/
[2] "Parallel Unix commands" - http://www-unix.mcs.anl.gov/sut/
[3] "Parallel print functions" - http://www.llnl.gov/sccd/lc/ppf/
[4] Picture from http://www.ptools.org/projects/mqm/flyer.html
[5] "Message Queue Manager project" - http://www.ptools.org/projects/mqm/
[6] "Scalable Unix Commands on Parallel Computers" - http://www-unix.mcs.anl.gov/sut/proposal.html
[7] "Unix Tools on Massively Parallel Processors" - http://www-unix.mcs.anl.gov/sut/long2/long2.html
[8] "Parallel print functions" - http://www.llnl.gov/sccd/lc/ppf/ppf_prospectus.html
[9] "Global Arrays" - http://www.emsl.pnl.gov:2080/docs/global/ga.html
[10] "The Condor Project Homepage" - http://www.cs.wisc.edu/condor/
[11] "Parallel Tools Consortium Projects" - http://www.ptools.org/projects.html
[12] "Cornell Theory Center. Parallel Tools" - http://www.tc.cornell.edu/UserDoc/Software/PTools/