NAME

mpipipe −− MPI−parallelized pipe for filter programs

SYNOPSIS

mpiexec [−n *nproc*] mpipipe [−n *record_count*] '*command* [*argument*...]'

DESCRIPTION

mpipipe invokes *command* with any supplied *argument* operands once for each set of contiguous input records read.

Parallelization

mpipipe uses a manager/worker model, with one manager process and *nproc*−1 worker processes.

Manager Process

The main loop of the manager process does the following:

1. Read *record_count* records from standard input.
2. Receive an output message from a worker process (the first such message will be empty). Send the input records from step 1 to the worker.
3. Print or store the output.

reads input from standard input, partitions the input records into sets of *record_count* contiguous records (the last such set possibly containing less than *record_count* records), waits for an output and sends each record set to a worker process.

Worker Processes

Each worker process executes a loop that does the following:

1. Send output of previous invocation of *command* to the manager process (the first such message will be empty).
2. Receive input from the manager process.
3. *fork()* a child process that invokes *command* on the input received in step 2. Store the output of this invocation.

Data Distribution

The size of each chunk distributed to the worker processes is determined mainly by the user with the **−n** option.

Load Balancing

The manager process handles load balancing by sending more input to a worker only after the worker has returned the output of the invocation of *command* on the previous input it received.

Scalability

The scalability of **mpipipe** depends on the user-specified *command*, *record_count*, and the cluster configuration. Regardless of these parameters, speed-up will eventually plateau as *nprocs* increases due to the bottleneck of having a single manager process; e.g., the manager process is unable to read, partition, and send input data to the worker processes fast enough to keep them from idling (due to disk I/O, CPU, or network bandwidth limitations).

Serial Optimization

While **mpipipe** can't improve the performance of *command*, there are a few things a user can do to minimize the overhead added by **mpipipe**:

- Ensure input comes from fast storage to prevent **mpipipe** from blocking while waiting for input.

- Don't use too-small values for *record_count*. If you have one million records, using the default *record_count* value of 1 will result in one million invocations of *command*!
- Don't use too-large values for *record_count* to ensure the input fits in memory, ensure every worker receives input to process, and allow load−balancing.
- If it is expected that processing times for any two same-sized record sets will be the same, the optimal value of *record_count* is **number of input records / number of workers processes**.

OPTIONS

−n *record_count*

The number of input records directed to the standard input of each invocation of *command*. In this prototype of **mpipipe**, a record is defined as a line.

The default value is 1.

OPERANDS

The following operand is supported:

'*command* [*−arguments...*]'

A command to execute. If the command contains arguments, operands, or I/O redirection operators, the entire command must be quoted.

The command should read its input from standard input and write its output to standard output, and there must be no data dependencies between

STDIN

Up to *record_count* contiguous records read from standard input will be used as standard input for each invocation of *command*.

There must be no data dependencies between any set of *record_count* **records.**

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

The interaction of MPI processes with signals is not specified by the MPI 2.1 standard; see *mpiexec* (1) for information on your implementation.

STDOUT

Standard output will be the standard output of each invocation of *command* on up to *record_count* contiguous records read from the standard input of **mpipipe**. The output of **mpipipe** −n *record_count command* will be the same as if *command* were invoked repeatedly, each invocation reading up to *record_count* records.

STDERR

The standard error will only be used in the case of fatal errors.

OUTPUT FILES

None.

EXIT STATUS

0 Execution completed successfully

>0 An error occurred

APPLICATION USAGE

mpipipe is recommended for trivially-parallizable problems where the input data consists of ''records'' of text, and the program to execute on the data is a filter — that is, it reads its input from standard input and writes its output to standard output.

EXAMPLES

1. For each line in the file ‘‘input.txt’’, print the hostname that the worker is executing on and the second column of the input line. Use two worker processes, and send two lines at a time to each worker (note that with an odd number of lines, the manager can only send one line to the worker that will process the final invocation of the user-specified command).

```
$ cat input.txt
1.1 1.2 1.3
2.1 2.2 2.3
3.1 3.2 3.3
4.1 4.2 4.3
5.1 5.2 5.3
6.1 6.2 6.3
7.1 7.2 7.3
8.1 8.2 8.3
9.1 9.2 9.3
$ mpiexec -n 3 mpipipe -n 2 \
"awk 'BEGIN { \"uname -n\" | getline hostname }
           {print hostname, \$2}'" < input.txt
hpc-class-00.ait.iastate.edu 1.2
hpc-class-00.ait.iastate.edu 2.2
hpc-class-34.ait.iastate.edu 3.2
hpc-class-34.ait.iastate.edu 4.2
hpc-class-00.ait.iastate.edu 5.2
hpc-class-00.ait.iastate.edu 6.2
hpc-class-34.ait.iastate.edu 7.2
hpc-class-34.ait.iastate.edu 8.2
hpc-class-00.ait.iastate.edu 9.2
```

NOTE:

The previous example as executed on hpc-class had to be wrapped in a script and submitted via qsub.

BUGS

- A incremental tag value is assigned to each set of *record_count* records, passed to the worker process in an *MPI_Send()*, and received from the worker process with an *MPI_Recv()*. This tag value is used by the manager process to order the output received from the workers so "mpipipe −n *record_count* ’*command*’" behaves the same as serial invocations of *command* with −*record_count* input records would. The MPI 2.1 standard guarantees only that the maximum tag value, indicated by the MPI_TAG_UB attribute, is at least 32,767, so it’s possible for a large amount of input to overflow this value if small *record_count* is used.
- OpenMPI implements blocking MPI routines using polling; this ensures low latency at the expense of CPU overhead. Depending on −*command* and −n *record_count*, the manager and worker processes could spend time significant time in a blocking MPI send or receive, thus use significant CPU resources. Where possible, a workaround for this could be to use non-blocking routines in a loop in conjunction with *sleep* (3) (or the higher-resolution *nanosleep* (3) on supported systems) to briefly suspend execution of the process before looping again; e.g.

```
bool received = false;
MPI_Irecv(...,request,...);
do {
    MPI_Test(request, received,...);
    if (received == true) break;
    else sleep(seconds);
} while (true);
```

NOTE:

The excessive CPU usage of blocking routines (specifically *MPI_Probe()*) was discovered first by using *top* (1) to identify that **mpipipe** was using excessive CPU resources, then by attaching to the running **mpipipe** process with Instruments (http://www.apple.com/macosx/developertools/instruments.html), which is Apple's front-end GUI to Sun Microsystems' DTrace, and identifying the function responsible for the excessive CPU usage.

- A hash table by *hcreate* (3) is used by the manager process to index the buffered worker output. This hash table has a fixed size, and could overflow given enough output.

NOTES

mpipipe requires an MPI implementation that can read data from standard input. Currently, only OpenMPI 1.2.3 or greater is supported.

SEE ALSO

mpiexec (1), *fork* (2), *hcreate* (3), *MPI_Abort* (3), *MPI_Alloc_mem* (3), *MPI_Comm_get_attr* (3), *MPI_Comm_rank* (3), *MPI_Finalize* (3), *MPI_Free_mem* (3), *MPI_Get_count* (3), *MPI_Init* (3), *MPI_Probe* (3), *MPI_Recv* (3), *MPI_Send* (3), *MPI_Sendrecv* (3), *OpenMPI* (3), *pipe* (2), *popen* (3),

The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition

Advanced Programming in the UNIX Environment, Second Edition, by W. Richard Stevens and Stephen A. Rago, published by Addison-Wesley Publishing Company, 2005.

MPI — The Complete Reference, Volume 1, The MPI Core, Second Edition, by M. Snir, S. Otto, S. Huss–Lederman, D. Walker, and J. Dongarra, published by MIT Press, 2000.

MPI: A Message-Passing Interface Standard, Version 2.1, by the Message Passing Interface Forum, published 2008.

AUTHOR

Nathan Weeks <*weeks@iastate.edu*>