COSC 4397
Parallel Computation

Debugging MPI applications and
1st Homework assignment

Edgar Gabriel
Spring 2010

Performance Metrics (I)

- **Speedup**: how much faster does a problem run on \( p \) processors compared to 1 processor?
  \[
  S(p) = \frac{T_{\text{total}}(1)}{T_{\text{total}}(p)}
  \]
  - Optimal: \( S(p) = p \) (linear speedup)

- **Parallel Efficiency**: Speedup normalized by the number of processors
  \[
  E(p) = \frac{S(p)}{p}
  \]
  - Optimal: \( E(p) = 1.0 \)
Performance Metrics (II)

• Example: Application A takes 35 min. on a single processor, 27 on two processors and 18 on 4 processors.

\[
S(2) = \frac{35}{27} = 1.29 \\
E(2) = \frac{1.29}{2} = 0.645 \\
S(4) = \frac{35}{18} = 1.94 \\
E(4) = \frac{1.94}{4} = 0.485
\]

Amdahl’s Law (I)

• Basic idea: most applications have a (small) sequential fraction, which limits the speedup

\[
T_{\text{total}} = T_{\text{sequential}} + T_{\text{parallel}} = fT_{\text{Total}} + (1-f)T_{\text{Total}}
\]

\[f: \text{fraction of the code which can only be executed sequentially}\]

\[
S(p) = \frac{T_{\text{total}}(1)}{(f + \frac{1-f}{p})T_{\text{total}}(1)} = \frac{1}{f + \frac{1-f}{p}}
\]
Example for Amdahl’s Law

Amdahl’s Law (II)

- Amdahl’s Law assumes, that the problem size is constant
- In most applications, the sequential part is independent of the problem size, while the part which can be executed in parallel is not.
Performance Metrics (III)

- **Scaleup**: ratio of the execution time of a problem of size $n$ on 1 processor to the execution time of the same problem of size $n^p$ on $p$ processors

$$S_p(n) = \frac{T_{\text{total}}(1,n)}{T_{\text{total}}(n^p,n^p)}$$

- Optimally, execution time remains constant, e.g.

$$T_{\text{total}}(p,n) = T_{\text{total}}(2p,2n)$$

Timing functions in MPI (I)

- Can be done e.g. by `gettimeofday()`
- **MPI functions provided**:

```c
double MPI_Wtime (void);
double MPI_Wtick (void);
```

- **MPI_Wtime** returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.
  - The times returned are local to the node that called them. There is no requirement that different nodes return `the same time`.
- **MPI_Wtick**: returns the resolution of MPI_WTIME in seconds.
Timing functions in MPI (II)

double starttime, endtime, elapsedtime;
...
starttime = MPI_Wtime();
/* do some incredibly complex calculations */
endtime = MPI_Wtime();
elapsedtime = endtime - starttime;

- Timing rules:
  - Make sure you time longer than the clock resolution (e.g. on a regular LINUX box clock resolution is ~10ms)
  - Rule of thumb: >100 times the clock resolution

Debugging sequential applications

- Several ways how to debug a sequential application:
  - printf() statements in the source code
    - Works, works reliably, painful to remove afterwards
  - assert() statements
    - check for a certain value of a variable. If the expression is false, the application aborts.
    - only active, if the macro NDEBUG is defined
      - Setting in the source #define NDEBUG 1
      - Compiling with the flag -DNDEBUG=1

#include <assert.h>

void open_record(char *record_name)
{
  assert (record_name!=NULL);
}
Using a debugger

- For a source file to be visible in the debugger, you have to compile the source code with the -g option, e.g.
  
  ```
  gabriel@salmon> mpicc -g -o test test.c
  ```

  - Avoid using optimization flags, such as -O3 when you would like to debug the code

- Two types of debugger
  - Command line debugger, such as `gdb`
  - Graphical debuggers, such as `ddd` (which is a GUI to `gdb`)

Load application into the debugger

Start app.

Show source code of app.

Show the value of a variable when the problem occurred.

Debugger points to the problem.
**gdb commands**

- Setting breakpoints: debugger stops execution at the specified line. Example
  
  ```
  (gdb) break errexample.c:10
  (gdb) break myfunc
  ```

- Stepping through the source code
  
  ```
  (gdb) next    (skips subroutines/functions)
  ```

- Continue execution (not step by step anymore)
  
  ```
  (gdb) cont
  ```

- Quit debugger
  
  ```
  (gdb) quit
  ```
Debugging a parallel application

- Some debuggers for parallel applications available (e.g. totalview, ddt)
  - Unfortunately expensive products
- You can still use printf and assert
  - Output from several processes will be mixed
  - you should put the rank of the process in front of each printf statement
- gdb or ddd still useable
  - You have to choose which process you would like to debug
  - Please be aware, that ddd or gdb can only see processes on the local machine

Debugging a parallel application (II)

- Hints for parallel debugging
  - Try to find the lowest number of processes for which the problem still occurs
  - Try to execute the application on a single node
    - If the problem does not show up on a single node, you will have to run the application on multiple nodes and login to the node, where the problem occurs
  - Introduce a sleep () statement in your application to have time to attach with a debugger
Attaching to a process

- **Menu File**
- **Bullet:** attach to processes
- **Choose the PID** which you would like to debug

### Debugging parallel applications (III)

- Some MPI libraries support the startup of a debugger in the `mpirun` command, including Open MPI

  ```bash
  mpirun -np 2 ddd ./colltest
  ```
  - only if all processes are running locally
  - Starts one `ddd` session per process
  - Not useful for large numbers of processes
1st Homework

- Rules
  - Each student should deliver
    - Source code (.c files)
    - Documentation (.pdf, .doc, .tex or .txt file)
      - explanations to the code
      - answers to questions
  - Deliver electronically to gabriel@cs.uh.edu
  - Expected by Monday, March 5th, 11.59pm
  - In case of questions: ask, ask, ask!
  - Max. number of points which can be achieved: 18 pts

Part A

Proc 0
--
MPI_Send(data, size, type, 1, 1, ...)
MPI_Recv(indata, size, type, 1, 1, ...)

Proc 1
--
MPI_Send(data, size, type, 0, 1, ...)
MPI_Recv(indata, size, type, 0, 1, ...)

- Above is a head to head send. This might or might not work depending on the system, the MPI implementation or other factors such as timing.
  - Write a paragraph on why the above is an incorrect (non-deterministic) MPI code. (2 Pts) Note: you will have to read probably chapter 3.5 of the MPI-1 specification!
  - Provide 2 different versions of the same example which correctly exchange data between two processes (hint: there are 4 simple ways) (4 Pts)

- Write a paragraph on the difference between blocking and non-blocking communication, and explain the usage and the constraints of non-blocking point-to-point operations. (2 Pts)
Part B: Calculating prime numbers

- A prime number is a positive integer evenly divisible by exactly two positive integers: 1 and itself. Sometimes, two consecutive odd numbers are both prime numbers (e.g. 5 and 7, or 11 and 13).

1. Write a parallel program to determine, for all integers less then 100,000,000 the number of times that two consecutive odd integers are both prime numbers. (6 pts total: 4 pts code, 2 pts design document)
2. Measure the execution time for the code on 1, 2, 4 and 8 processors. (2 pts)
3. Determine the parallel speedup and the parallel efficiency of your code on the shark cluster for the 2, 4, and 8 processor cases. (2 pts)

How to use a cluster

- A cluster usually consists of a front-end node and compute nodes
- You can login to the front end node using ssh (from windows or linux machines) using the login name and the password assigned to you.
- The front end node is supposed to be there for editing, and compiling - not for running jobs!
  - If 41 students would run their jobs on the same processor, everything would stall!!!!!
- To allocate a node for interactive development:
  
  ```
  teamxy@shark:~$ alloc -n 4 bash
  teamxy@shark:~$ squeue
  JOBID PARTITION  NAME USER ST TIME NODES NODELIST(REASON)
  489 calc smith R 0:02 2 shark[08-09]
  teamxy@shark:~$ mpirun -np 4 ./mytest
  ```
How to use a cluster (II)

- Note: the `mpirun` command will know where to execute the job only if you type the command from the very same terminal/window where you typed the `salloc` command.
- Maximum time you an allocation will be available: 30 minutes
  - after that time, your job gets automatically killed by the system
  - fairness rule
- Note, that when you request e.g. 8 processors, you might receive either
  - 4 nodes with 2 cores [shark01-24] or
  - 1 node with 8 cores [shark25-29]

Once your code is correct and you would like to do the measurements:
- You have to submit a batch job
- The command you need is `sbatch`, e.g.
  ```bash
  sbatch -n 8 ./measurements.sh
  ```
- Your job goes into a queue, and will be executed as soon as a node is available. You can check the status of your job with
  ```bash
  squeue
  ```
How to use a cluster (III)

- The output of `squeue` gives you a job-id for your job
- Once your job finishes, you will have a file called `slurm-<jobid>.out` in your home directory, which contains all the output of your `printf` statements etc.
- Note the batch script used for the job submission (e.g. `measurements.sh`) has to be executable. This means, that after you downloaded it from the webpage and copied it to shark, you have to type
  
  ```bash
  chmod +x measurements.sh
  ```

  - Please do not edit the `measurements.sh` file on MS Windows. Windows does not add the UNIX EOL markers, and this confuses `slurm` when reading the file.

Notes

- If you need hints on how to use a UNIX/Linux machine through ssh:
  
  [http://www.cs.uh.edu/~gabriel/cosc4397_s06/ParCo_08_IntroductionUNIX.pdf](http://www.cs.uh.edu/~gabriel/cosc4397_s06/ParCo_08_IntroductionUNIX.pdf)

- How to use a cluster such as shark
  
  [http://pstl.cs.uh.edu/resources.html](http://pstl.cs.uh.edu/resources.html)