COSC 6374
Parallel Computation

Parallel Measures and Debugging MPI application

Edgar Gabriel
Fall 2011

Performance Metrics (I)

- **Speedup**: how much faster does a problem run on \( p \) processors compared to 1 processor?

\[
S(p) = \frac{T_{total}(1)}{T_{total}(p)}
\]

- 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.

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

Amdahl’s Law (I)

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

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

\(f\): fraction of the code which can only be executed sequentially

\[
S(p) = \frac{T_{\text{total}}(1)}{(f + \frac{1-f}{p})T_{\text{total}}(1)} = \frac{1}{f + \frac{1-f}{p}}
\]
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

\[
S_c(p) = \frac{T_{total}(1,n)}{T_{total}(p,n*p)}
\]

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

\[
T_{total}(p,n) = T_{total}(2p,2n)
\]

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)

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

```c
#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.

Debugger points to the problem

Show the value of a variable when the problem occurred
**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)
  (gdb) step (enters 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 usable
  - 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
  - `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