Distributed and Parallel Technology

Parallel Performance Tuning

Hans-Wolfgang Loidl
http://www.macs.hw.ac.uk/~hwloidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

[0]No proprietary software has been used in producing these slides
[0]Based on earlier versions by Greg Michaelson and Patrick Maier

Techniques for Parallel Performance Tuning

To achieve good parallel performance, you need to

- know basics about the *sequential performance*
- predict the asymptotic *complexity of computation and communication*
- *measure* the performance of the application
- possibly, *restructure* the program to enhance parallel performance

Measuring Execution Time — Whole-program profiling

We want to find out: where is time taken in program?
`gprof` is a Linux profiling tool

```
# gcc -pg -o m3 matrix3.c
# ./m3 2MAT_2000_10_65536
2000 * 2000; SEQUENTIAL; 58.890000 secs
# gprof m3 gmon.out
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.16     53.47    53.47  4000000     0.00     0.00  dotProd
  0.16     53.56     0.09        1     0.09    53.56  matrixProd
  0.12     53.62     0.07        1     0.07     0.07  transpose
  0.02     53.63     0.01        2     0.01     0.01  readMatrix
  0.00     53.63     0.00     8000     0.00     0.00  allocVector
  0.00     53.63     0.00        4     0.00     0.00  allocMatrix
```


Measuring Execution Time — Cache profiling

We want to find out how efficiently the cache is used.
`cachegrind`, a component of the `valgrind` suite, gives this information.

```
# valgrind –tool=cachegrind m2 2MAT_2000_10_65536
```

```
2000 * 2000; SEQUENTIAL; 1452.630000 secs

 I   refs:      87,822,991,690
 I1  misses:             1,294
 L2i misses:             1,285
 I1  miss rate:           0.00%
 L2i miss rate:           0.00%

 D   refs:      51,234,059,144  (49,989,145,968 rd   + 1,244,913,176 wr)
 D1  misses:     9,508,333,358  ( 9,507,571,877 rd   +       761,481 wr)
 L2d misses:       502,269,132  (   501,509,253 rd   +       759,879 wr)
 D1  miss rate:           18.5% (          19.0%     +           0.0%  )
 L2d miss rate:            0.9% (           1.0%     +           0.0%  )

 L2 refs:        9,508,334,652  ( 9,507,573,171 rd   +       761,481 wr)
 L2 misses:        502,270,417  (   501,510,538 rd   +       759,879 wr)
 L2 miss rate:             0.3% (           0.3%     +           0.0%  )
```

Measuring Execution Time — Sequential C

```
#include <time.h>

...

/* global variable */
clock_t start_time;

...

/* start timer */
start_time = clock();

/* do something */
...

/* take time */
printf("Elapsed time: %f secs", (clock() - start_time) / CLOCKS_PER_SEC);
```

Measuring Execution Time — C with MPI

```
/* global variable */
double elapsed_time;

...

/* start timer */
MPI_Barrier(MPI_COMM_WORLD);
elapsed_time = - MPI_Wtime();

/* do something */
...

/* take time */
elapsed_time += MPI_Wtime();
printf("Elapsed time: %f secs", elapsed_time);
```

Useful MPI functions:

- `double MPI_Wtime(void)`
  - current time in seconds; use `MPI_Wtick()` to enquire precision
- `int MPI_Barrier(MPI_Comm comm)`
  - synchronizes all processors in communicator `comm`
  - use only for timing the whole system

What To Measure

Measure runtime of

- whole program
  - too pessimistic
  - system artefacts (eg., I/O, MPI startup) distort timings
- whole program *without* system artefacts
  - exclude time spent on I/O (or suppress I/O)
  - exclude time spent on MPI startup/shutdown
  - Do measure whole program, not only parallel parts!
- whole program *without* system artefacts and *without* inherently sequential parts
  - only measure performance of code that does run in parallel
  - disreputable — see Amdahl's Law

Speedup

Speedup is a measure of how much effect adding extra processors has on runtime.

- *absolute* speedup
  = sequential time on 1 processor / parallel time on n processors
- *relative* speedup
  = parallel time on 1 processor / parallel time on n processors
  - Typically, parallel execution on 1 processor is slightly slower than sequential execution due to MPI overhead.
  - Typically, absolute speedup $<$ relative speedup.

Speedup — Amdahl's Law

- Assumption: Parallel program can be divided into a *sequential part* (which must be executed by 1 processor only) and a *parallel part* (which may be executed by many processors).
- *Ideal runtime* on n processors $T_n = T_s + T_p/n$
  - T_s = execution time of sequential part
  - T_p = execution time of parallel part on 1 processor
- *Ideal speedup* on n processors = $\frac{T_s+T_p}{T_s+T_p/n}$

Amdahl's Law

Let $f = T_s/(T_s + T_p) = T_s/T_1$ be the *sequential fraction* of a program. The maximum achievable speedup on n processors is bounded by

$$\frac{1}{f + (1-f)/n}$$

Corollary: For $n \to \infty$, speedups are bounded by $1/f = T_1/T_s$.

Measuring Parallel Runtime — `matrix8.c`

```
int main(int argc, char ** argv)
{
  /* timers for profiling */
  double t_total, t_sort, t_merge_comm, t_merge_comp;
  ...
  MPI_Init(&argc, &argv);  MPI_Comm_size(MPI_COMM_WORLD, &p);  MPI_Comm_rank(MPI_
  if (id == 0) { /* master */
    ...
    /* start the timer */
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = - MPI_Wtime();
    ...
    int ** M2T = transpose(M2, n, m);
    int ** M3 = matrixProdMaster(M1, M2T, m, n, p-1);
    /* stop the timer */
    elapsed_time += MPI_Wtime();
    /*  write matrix to /dev/null */
    FILE * fout = fopen("/dev/null", "w");
    writeMatrix(fout, M3, m, m);
    printf("%d * %d; %2d processors; %f secs", m,n,p,elapsed_time);
  }  else { /* worker */
    ...
    matrixProdWorker(m, n, p-1, id);
  }
  return 0;
}
```

Tutorial: Parallel Matrix Multiplication — Setup

```
# download the sequential as baseline for comparison
> wget -q http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/matrix3.c
# download input data
> wget -q http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/2MAT_2000_10_65536
# compile sequential version (with optimisation!)
> gcc -Wall -O -o matrix3 matrix3.c
# run sequential version
> ./matrix3 2MAT_2000_10_65536
2000 * 2000; SEQUENTIAL; 12.240000 secs

# download parallel version
> wget -q http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/matrix8.c
# compile parallel version
> mpicc -Wall -O -o matrix8 matrix8.c
# run it on 1 worker (and 1 master)
```

Tutorial: Parallel Matrix Multiplication — Sample Runs

```
# sequential run
> ./matrix3 2MAT_2000_10_65536
2000 * 2000; SEQUENTIAL; 12.240000 secs

# now you can run the parallel MPI program, using 1 worker and 1 master
> mpirun -np 2 matrix8 2MAT_2000_10_65536
2000 * 2000;  2 processors; 12.554089 secs
# run it on 2 workers
> mpirun -np 3 matrix8 2MAT_2000_10_65536
2000 * 2000;  3 processors; 6.468304 secs
# NB: speedup of almost 2, good!
# run it on 4 workers
> mpirun -np 5 matrix8 2MAT_2000_10_65536
2000 * 2000;  5 processors; 3.260149 secs
# run it on a network of 4 nodes (listed in mpi4) with 40 workers in total
> mpirun -np 40 -f mpi4 matrix8 2MAT_2000_10_65536
2000 * 2000; 40 processors; 5.672003 secs
```

Tutorial: Parallel Matrix Multiplication — Sample Runs

Batch job of executions on 1 to 16 workers (put this into a shell script!):

```
> echo "Workers 1">LOG ; ./matrix3 2MAT\_2000\_10\_65536 >> LOG ; for ((i=1;i
> cat LOG | sed -e '/secs/a\X' | sed -e 's/^.*np \([0-9]*\).*$/\1/;/PEs/d;s/^
> cat LOG | sed -e '/secs/a\X' | sed -e 's/^.*np \([0-9]*\).*$/\1/;/PEs/d;s/^
> echo "set term x11; plot \"rt.dat\" with lines; pause 10 " | gnuplot
```

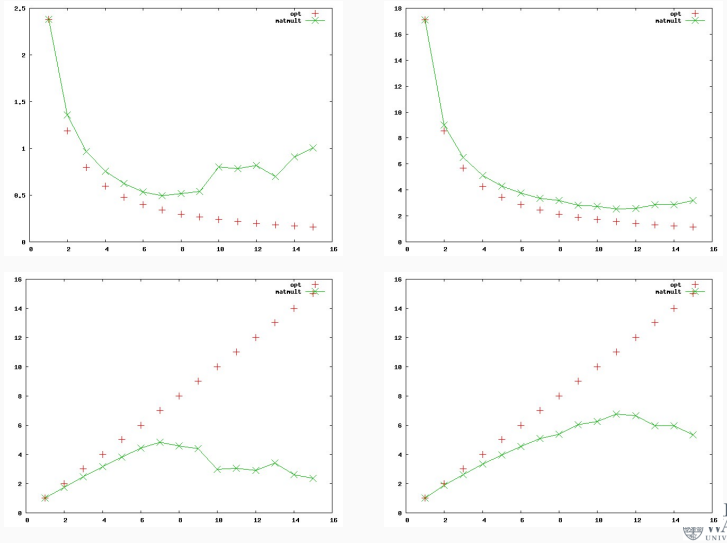
Check the F21DP FAQ page for details: `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/faq.html`

Parallel Performance — Matrix Multiplication

1000×1000 2000×2000

runtime [s]

rel speedup



Parallel Performance — Matrix Multiplication

Observations:

- good "near linear" speedups only up to 6–8 processors
- speedups peak and then decline
- larger problem size delays peak

Analysis:

- Processing cost per processor decreases linearly with #processors.
- Communication cost per processor is roughly independent of #processors.
  - This statement is only true for workers (but they dominate).
  - Communication cost of worker dominated by cost of receiving full matrix $M2$.
- More processors $\rightsquigarrow$ *decreasing ratio computation/communication*
- Beyond peak speedup, communication overwhelms processing.

Some Profiling Tips

How to instrument your code:

- Total parallel runtime: Start timer after a barrier.
- Communication: Time receives (including collectives).
  - Includes time for synchronisation. A barrier before receive will measure only communication but will massively distort algorithm.
- Other program parts: Avoid distorting timing by I/O or comm.
  - Postpone I/O or measure and subtract I/O time.
  - Printing profile is I/O: Do it at the end (after a barrier).
  - Don't comment out I/O! Compiler may optimise your program away.
- **Note:** Instrumenting changes your code (cache behaviour, ...)

How to run your experiments:

- Profile on lightly loaded machines.
  - Check load on nodes (eg. `uptime`) before running experiments.
  - Don't use many nodes for a long time.
- Take the median of several profiles (for a given problem size).
  - Fix some random input (if any) of the given problem size.
  - Pick the profile with median *total runtime*.
    - ★ Don't mix data across runs and don't average.