INF5380/INF9380 NORBIS course
High-performance computing in bioinformatics

Introduction to parallel programming

Torbjørn Rognes
torognes@ifi.uio.no

Department of Informatics, UiO
20 April 2016

UiO : Universitetet i Oslo

PART 1: INTRODUCTION TO PARALLEL PROGRAMMING

The need for parallelisation

- We want to process ever larger amounts of data
- We want to do it in reasonable time
- The processing power of a single CPU core is not increasing very much any more; CPU frequencies have almost stopped at about 3GHz
- Moore’s law about the doubling of the number of transistors every 18 months still apply, but mainly results in more cores instead of faster cores.
- Parallel programming is required to tap the potential of modern computers.

Hardware enabling parallel execution

- Grid computing, multiple individual computer systems (e.g. Folding @ Home)
  - Communicates over Internet
- Multiple computers in one cluster (e.g. Abel)
  - Usually a fast network between them
- Multiple cpus in one computer (e.g. a server)
  - shared memory
  - several ordinary cpus
  - graphics processing units (gpus) or other co-processors
- Multiple cores in one cpu (e.g. most laptops)
  - 2, 4, 8 or even more cores is now common in many cpus
- Multiple execution units in one core
  - Two or more logical/arithmetic units
  - Hyperthreading (HT)
- Bit-level parallelism = SIMD within a register (SWAR)
  - Wide registers (e.g. 64 bits) that can be divided into smaller units (e.g. 8 bits) that can be operated on independently

Definition of serial and parallel systems

- We are considering basic operations like:
  - add two numbers (or other mathematical/logical operations)
  - compare two numbers
  - retrieve a number from memory
  - store a number in memory
  - jump to somewhere else in the program (if some condition)

- A serial system can only perform one of these operations per unit of time

- A parallel system can perform two or more of these simultaneously

A serial vs a parallel program

- A problem may involve N separate independent steps
- If we have one processing cores available, this problem can be solved in N units of time
- If we have k processing cores available, the problem may ideally be solved in N/k units of time
- If k>=N, the problem may be solved in 1 unit of time

Parallelisation may be easy or hard

Easy problems:

- Some problems are easy to parallelise
- "Embarrassingly parallel"
- Problem may be divided into parts that can be solved independently.
- No communication between processes needed.
- Example: Searching multiple queries against a database. Each search is totally independent of the others.

Hard problems:

- Other problems are hard or impossible to parallelise
- Difficult or impossible to divide the problem into independent parts. E.g. each step depends on the previous step.
- Extensive communication between processed required
- Example: Some cryptographic schemes are designed to be hard to parallelise.

Example 1: Easy

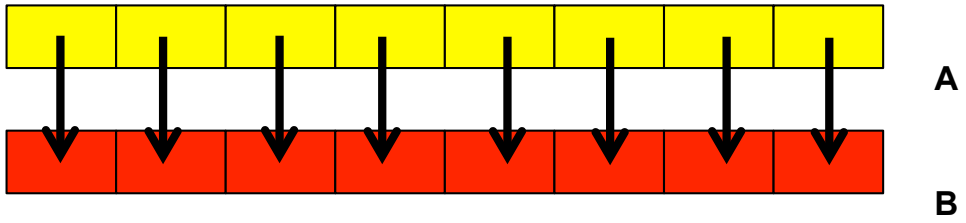
```
/* compute b[i]=2*a[i] for i=0 to N-1 */

#define N 8
int a[N] = { 94, 43, 48, 58, 1, 27, 28, 53 };
int b[N];

for(int i=0; i<N; i++)
{
        /* independent */
        b[i] = 2 * a[i];
}
```

Example 1: Difficult

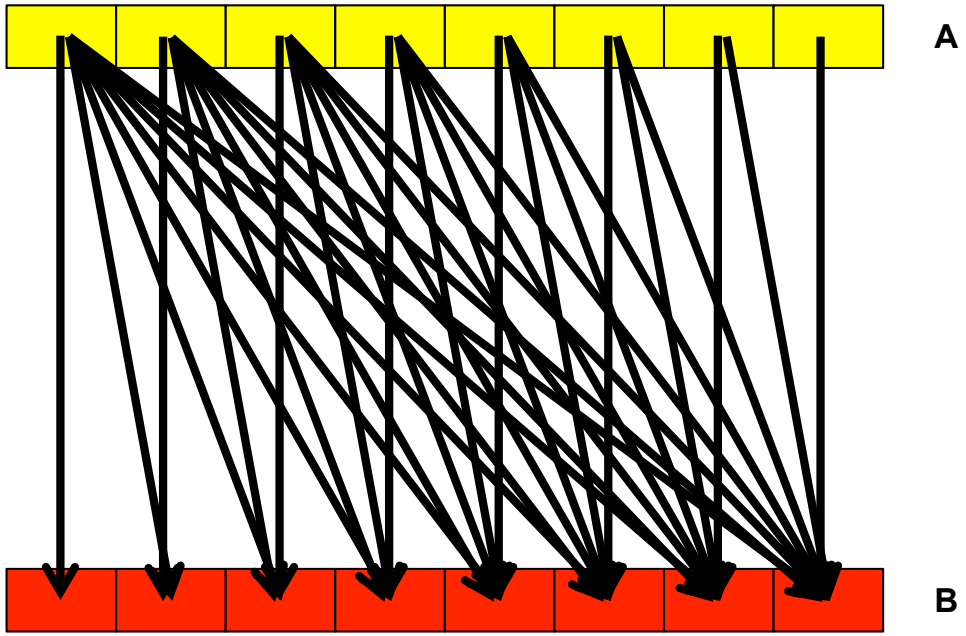
```
/* compute b[i]=sum(a[0], a[1], ..., a[i]) for i=0 to N-1*/

#define N 8
int a[N] = { 94, 43, 48, 58, 1, 27, 28, 53 };
int b[N];

b[0] = a[0];

for(int i=1; i<N; i++)
{
        /* dependent on previous value */
        b[i] = b[i-1] + a[i];
}
```

Data dependencies

Easy case



Difficult case



Two main parallel computer models

- Distributed-memory systems: processes communicate over a network
  - grid
  - cluster

- Shared-memory systems: processes communicate through shared memory
  - single machine
  - multiple cpus, cores, execution units

Distributed-memory models

- Computers in a cluster with separate memory communicating by sending messages to each other over a network
- Communication may be direct node to node or using broadcasting (one to many)
- MPI = Message Passing Interface
- openmpi
- **MapReduce** is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.

Shared-memory models

A Parallel Random-Access Machine (PRAM) is a model of a shared-memory parallel computer. Four categories are defined:

- **EREW**: Exclusive Read and Exclusive Write
- **ERCW**: Exclusive Read and Concurrent Write
- **CREW**: Concurrent Read and Exclusive Write
- **CRCW**: Concurrent Read and Concurrent Write

Notes:

- **Exclusive** read or write means that only one process may read or write to the same memory simultaneously.
- **Concurrent** read or write means that only one process may read or write to the same memory simultaneously.
- With **concurrent write**, it is necessary to define the result of the operation (arbitrary, based on rank, sum/max, ...)

Coding on shared-memory architectures

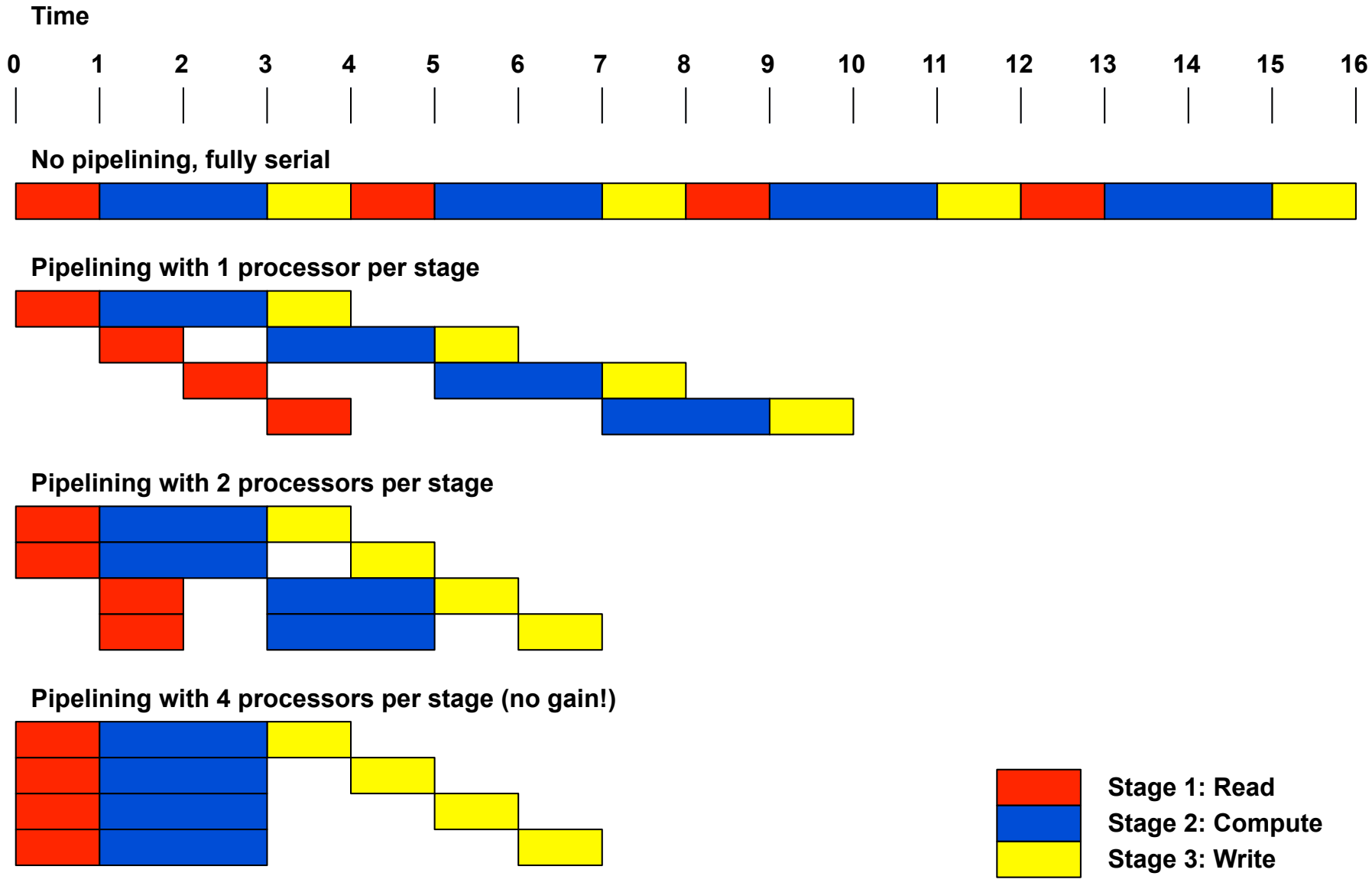
- OpenMP: We give hints to the compiler about parts of the code that may be parallelised, but leave most of the job to the compiler.

- pthreads: We manage creation and termination of separate threads ourselves and take care of synchronisation.

Pipelining

- We have a problem that requires, for instance, 3 sequential stages of operations on each data value:
  1. read (red)
  2. compute (blue)
  3. write (yellow)
- We cannot write data simultaneously
- We can have multiple (N) execution units that perform the read and compute stages.
- How can we do this quickly?
- Use pipelining!

Pipelining example



Flynn's taxonomy

SISD: Single Instruction stream, Single Data stream

- Perform one operation on one value (traditional)

SIMD: Single Instruction stream, Multiple Data stream

- Perform the same operation on many different values simultaneously (GPU, SIMD-instructions, vectorisation)

MISD: Multiple Instruction stream, Single Data stream

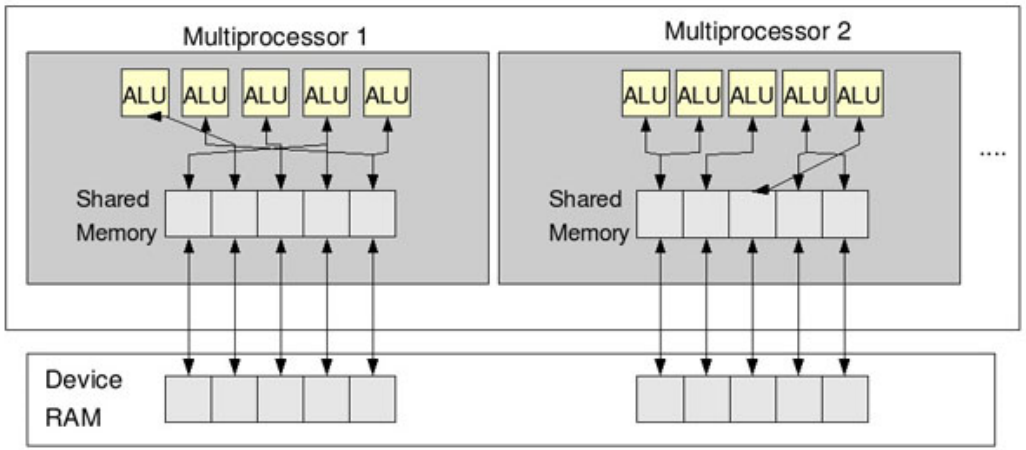
- Perform many different operations on the same value simultaneously (Uncommon, for fault detection)

MIMD: Multiple Instruction stream, Multiple Data stream

- Perform many different operations on many different values simultaneously (multiple core, distributed systems)

Graphics Processing Units (GPUs)

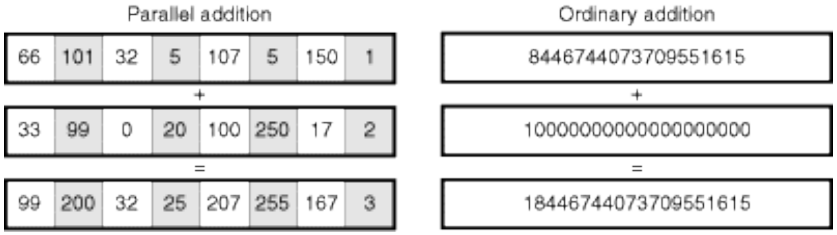
- Originally made for games, now also for high-performance computing
- Manufacturers: Nvidia, ATI and others
- Up to about 5000 cores total
- Divided into X multiprocessors, each with Y stream processors
- Relatively simple processors
- All stream processors within one multiprocessor execute the same code in sync.



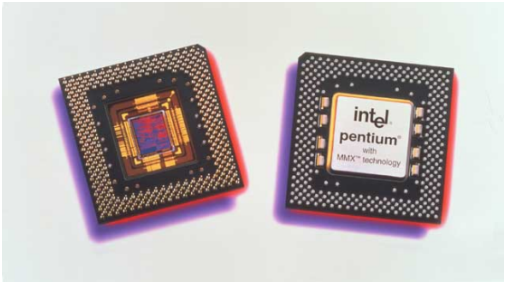
Sources: Nvidia, ATI

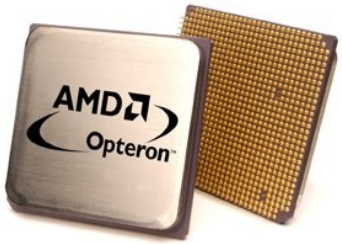
SIMD technology in ordinary CPUs

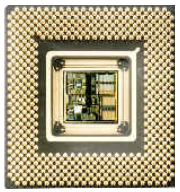
- SIMD = Single-Instruction Multiple Data
- Perform the same operation in parallel on multiple data streams
- Kind of vector processing
- Used for high-speed processing of various media like images, sound, video, as well as cryptography
- Allows up to 16 simple operations to be executed in parallel



- Embedded in most modern microprocessors







Sources: Intel, AMD, IBM

Writing SIMD code

- Assembler

```
__asm__ (”psubsb xmm15, xmm12”);
```

- SIMD Intrinsics

```
#include <tmmintrin.h>
E = _mm_subs_epi8(E, R);
```

- C (with optimizing compiler)

```
for(int i=0; i<16; i++)
    E[i] = E[i] – R[i];
```

Popular parallel programming model

- The master/slave or parent/child model
- One main process (master or parent) spawns one or more child processes (slaves or children)
- The master process usually:
  - handles user input
  - creates the child processes
  - allocates work to each of the child processes
  - collects the results from the child processes
  - ensures all children are finished
  - handles user output
- The slave process usually:
  - waits for work to become available
  - carries out the work
  - submits results
  - loops until asked to terminate or no more work is available
  - terminates

Creating a new process or thread

- Create a new process:

```
int fork(void)
```

Returns the process ID of the new child process to the parent, while the child process receives the value 0. Both the parent and the child now executes in parallel from the same starting point, identical data etc.

- Create a new thread:

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg)
```

Starts a new thread that executes the start_routine function.

Waiting for a child to finish

- Wait until any child process has finished:

```
pid_t wait4(pid_t pid,
            int *stat_loc,
            int options,
      struct rusage *rusage)
```

  Waits (pauses the calling process) until a given child process with given process id has terminated. Returns information about resource usage.

- Wait until a given child thread has finished:

```
int pthread_join(pthread_t thread,
                 void **value_ptr)
```

  Waits (pauses the calling thread) until the given child thread has terminated. Returns result value from child thread.

Thread synchronisation

- Synchronisation is often necessary for correct operation in parallel systems.
- Critical sections need to be protected.
- If two or more threads need to update a common variable they need to do it in an orderly manner.
- We need to guarantee mutual exclusion of certain events.
- Mechanisms:
  - lock
  - monitor
  - semaphore
  - condition variable
  - mutex
  - ...
- Requires hardware support for certain atomic operations

Example without synchronisation

- Example of withdrawal of money from bank account without proper synchronisation.
- Two different invoices of $100 and $200 are to be paid from a bank account.

Thread 1	Thread 2	Account
Read amount: $1000		$1000
	Read amount: $1000	$1000
Deduct $200: $800		$1000
	Deduct $100: $900	$1000
Write amount: $800		$800
	Write amount: $900	$900

Example with synchronisation

- With proper synchronisation using a lock
- Critical sections are indicated

Thread 1	Thread 2	Account
Acquire lock	Acquire lock	$1000
Lock it	Wait	$1000
Read amount: $1000	Wait	$1000
Deduct $200: $800	Wait	$1000
Write amount: $800	Wait	$800
Unlock it	Wait	$800
...	Lock it	$800
...	Read amount: $800	$800
...	Deduct $100: $700	$800
...	Write amount: $700	$700
...	Unlock it	$700

PART 2: ANALYSING CODE

Comparing serial and parallel programs

- When we compare the time used by a parallel program with a serial program that performs the same task, it is important that we do a fair comparison.

- We must compare the parallel program to the best available implementation of the serial program.

Speedup

The speedup of a parallel vs a serial program is defined as

$S_{latency} = L_{serial} / L_{parallel}$

where

L_{serial} = time used for serial program

and

$L_{parallel}$ = time used for parallel program

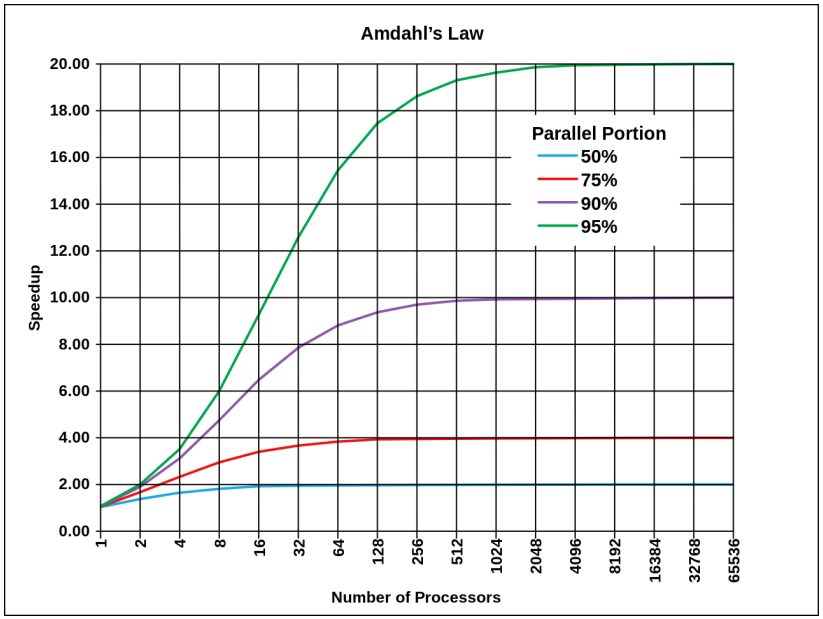
Linear and super-linear speedup

- Linear speedup: When the speedup is (almost) equal to the number of threads used

- Super-linear speedup: Special cases where the speedup is even higher than linear

Amdahl's law

$$S_{\text{latency}}(s) = \frac{1}{1 - p + \frac{p}{s}}$$



where

S_{latency} = speedup of whole task

s = speedup of the part of the task that can be parallelised

p = fraction of execution time of the serial program that is spent in the part that can be parallelised

Source: Wikipedia

Efficiency

Efficiency of utilization of computing resources is defined as

Efficiency = S / s

where

S = Speedup of the whole task

s = Speedup of the part of the task that can be parallelised

Efficiency = 1 means perfect utilization of resources
Efficiency = 0 is very bad

What should we parallelise?

- Code that takes significant time (don’t bother with the rest)
- Code that is executed many times over and over... (inner loops)
- Operations that can be performed independently on different parts of the data (that are easy to parallelise)
- We can use code profiling to find out, or we can analyse manually

Where is the bottleneck?

It is not always the actual computations that is the bootleneck for performance.

Other factors:

- Memory access
  - Memory access may be a bottleneck
  - Non-linear access may be problematic
  - Optimal cache usage may be very important

- Disk access:
  - Non-local or slow disks
  - Non-linear access to disks may be slow

Compiler flags for optimization

–O0: Do not optimize. Reduce compilation time and make debugging produce the expected results. This is the default.

–O or –O1: Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function. With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

–O2: Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code.

–O3: Optimize yet more. -O3 turns on most optimizations.

Parallel execution in the shell

- To run a command in the background in the shell, just follow it with an ampersand:

```
command &
```

- Multiple commands can be started and run in parallel:

```
first & second & third & fourth &
```

- Or with a for loop and perhaps with the help of seq:

```
for n in $(seq 1 8); do
      ( echo starting thread $n ; sleep $n ; Finished thread $n ) &
done
```

Controlling and monitoring processes

Monitoring processes:

- top, ps (unix)
- Task manager (Windows)
- Activity monitor (Mac OS X)

Stopping/Pause processes:

- kill
- Ctrl-C
- Ctrl-Z

Background / foreground jobs

- fg
- bg
- jobs

Threads and processes

- Threads are “lightweight” processes running within the same process
- Less overhead in terms of memory and time when starting a new thread compared to a process

Important computing resources

- time
  - user: time used inside the program (sum of all processes)
  - system: time used by the operating system (e.g. reading files) (sum of all processes)
  - wall: actual, real time used ("wall clock time")

- memory
  - huge nodes (1TB)
  - most nodes 64GB
  - average 4GB per core on most Abel nodes

- i/o: input/output
  - disk read/write
  - local disk, remote disk etc
  - linear / random access

Timing

- **`/usr/bin/time`** gives
  - real/elapsed (wall) time
  - user time
  - system time
- It can also give information about memory usage and pages swapped

/usr/bin/time examples

```
$ /usr/bin/time tool1
28.03user 0.23system 0:28.27elapsed 99%CPU (0avgtext+0avgdata 416maxresident)k
0inputs+0outputs (0major+127minor)pagefaults 0swaps
```

```
$ /usr/bin/time tool2
30.88user 0.00system 0:04.07elapsed 758%CPU (0avgtext+0avgdata 644maxresident)k
0inputs+0outputs (0major+191minor)pagefaults 0swaps
```

```
$ /usr/bin/time sleep 10
0.00user 0.00system 0:10.00elapsed 0%CPU (0avgtext+0avgdata 600maxresident)k
0inputs+0outputs (0major+182minor)pagefaults 0swaps
```

Alternative timing

Print or read out the time from within the program

In C, the number of seconds since 1 Jan 1970 is returned by

```
time(0)
```

More fine-grained resolution available with gettimeofday():

```
long getusec(void)
{
  struct timeval tv;
  if(gettimeofday(&tv,0) != 0) return 0;
  return tv.tv_sec * 1000000 + tv.tv_usec;
}
```

Memory use reported by /usr/bin/time

If you run **/usr/bin/time** with a command, it will report some statistics on the memory usage:

```
$ /usr/bin/time tool
27.63user 0.04system 0:27.67elapsed 100%CPU (0avgtext+0avgdata 420maxresident)k
0inputs+0outputs (0major+128minor)pagefaults 0swaps
```

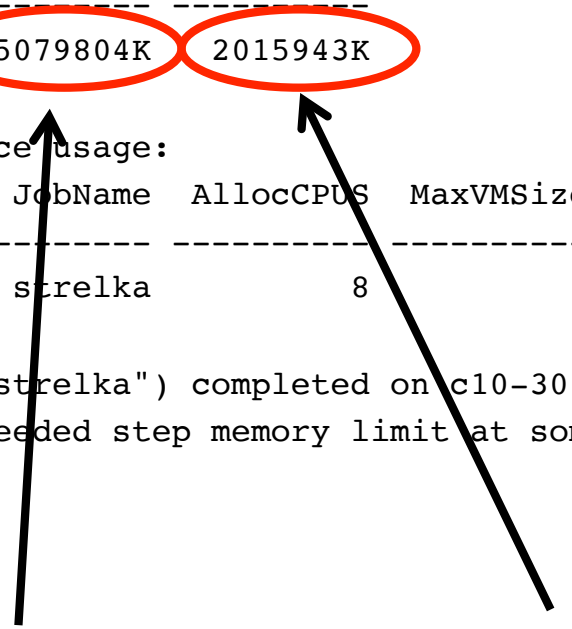
maxresident = 420k

The maximum resident set size of the process during its lifetime, in Kbytes.

This is the maximum amount of real memory used.

Slurm report when job finished

```
No Modulefiles Currently Loaded.

Job script resource usage:
       JobID  MaxVMSize     MaxRSS
------------ ---------- ----------
14175952.ba+   5079804K   2015943K

Job step resource usage:
       JobID    JobName  AllocCPUS  MaxVMSize     MaxRSS    Elapsed ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
14175952        strelka          8                          02:40:46      0:0

Job 14175952 ("strelka") completed on c10-30 at Wed Mar 2 19:04:42 CET 2016
slurmstepd: Exceeded step memory limit at some point.
```



Maximum memory usage, virtual memory: about 5GB

Maximum memory usage, resident set size (rss): about 2GB

Your program will be terminated if it uses more than allocated

Profiling

- Profiling can tell you how much time is spent on which parts of your program
- It helps you to finding the parts of the program worth spending an effort to optimize or parallelise

gprof

- Compile and link your program with gcc using the -pg and -g options:

```
gcc -g -pg -o mytool mytool.c
```

- Run your code:

```
./mytool
```

- Run gprof to see results:

```
gprof ./mytool | more
```

Example gprof report

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.34	0.02	0.02	7208	0.00	0.00	open
16.67	0.03	0.01	244	0.04	0.12	offtime
16.67	0.04	0.01	8	1.25	1.25	memccpy
16.67	0.05	0.01	7	1.43	1.43	write
16.67	0.06	0.01				mcount
0.00	0.06	0.00	236	0.00	0.00	tzset
0.00	0.06	0.00	192	0.00	0.00	tolower
0.00	0.06	0.00	47	0.00	0.00	strlen
0.00	0.06	0.00	45	0.00	0.00	strchr
0.00	0.06	0.00	1	0.00	50.00	main
0.00	0.06	0.00	1	0.00	0.00	memcpy
0.00	0.06	0.00	1	0.00	10.11	print
0.00	0.06	0.00	1	0.00	0.00	profil
0.00	0.06	0.00	1	0.00	50.00	report

gprof report numbers

- **% time:** This is the percentage of the total execution time your program spent in this function. These should all add up to 100%.
- **cumulative seconds:** This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.
- **self seconds:** This is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number.
- **calls:** This is the total number of times the function was called. If the function was never called, or the number of times it was called cannot be determined (probably because the function was not compiled with profiling enabled), the calls field is blank.
- **self ms/call:** This represents the average number of milliseconds spent in this function per call, if this function is profiled. Otherwise, this field is blank for this function.
- **total ms/call:** This represents the average number of milliseconds spent in this function and its descendants per call, if this function is profiled. Otherwise, this field is blank for this function. This is the only field in the flat profile that uses call graph analysis.
- **name:** This is the name of the function. The flat profile is sorted by this field alphabetically after the self seconds and calls fields are sorted.

PART 3: HANDS-ON EXERCISES

Assignment

- We will use a program that generates random sequences and then measures the length of the longest homopolymer in the sequence, e.g. the longest stretch of the same base

  AAGTGCTAGTCATA**TTTTTTTTTT**ACGTGTATGTGT

  Here, there is a stretch of 10 T's.

- There is a serial and a parallel variant of the program.

- We will try the following:
  - profiling the code
  - testing compiler optimizations
  - measure the time used by the serial and the parallel variant
  - measure the speedup when using 1-8 parallel threads
  - monitoring the processes

Start interactive session

```
ssh abel.uio.no

qlogin --account=ln0002k --ntasks-per-node=8 --mem-per-cpu=3G

git clone https://github.com/torognes/inf9380

cd inf9380/parallel-programming/src

make

/usr/bin/time ./serial
```

Monitor the running program

- Find the cluster node you are logged into:

```
uname –n
```

- Log into that node (e.g. c16-27) from a new window:

```
ssh c16-27
```

- Then start monitoring:

```
top
```

- Look for the program “serial” and check %CPU.

Profile the program

- Compile with the –g and –pg options (modify Makefile), but disable optimizations (no –O2 or similar):

```
g++ –o serial serial.c –Wall –g –pg
```

- Run the program (might take a bit longer):

```
/usr/bin/time ./serial
```

- Check the output:

```
./gprof ./serial | more
```

Turn on optimizations

- Compile with the –O1, -O2, or –O3 compiler options (modify Makefile):

```
gcc –o serial serial.c –Wall –O2 –g
```

- Run the program and measure time used:

```
/usr/bin/time ./serial
```

- Compare the time used with no or the three different levels of optimization.

- Did optimization help?

Increase number of threads

- Modify the source code in parallel.c to increase the number of threads used from 1 to 2, 3, 4, 5, 6, 7 and 8 (but not higher):

```
#define THREADS 8
```

- Compile the program after any change, with optimizations:

```
make
```

- Time and run the program:

```
/usr/bin/time ./parallel
```

Performance and speedup

- Plot the time used vs the number of threads
- For any number of threads 1-8 calculate the speedup relative to the serial program, and the efficiency
- How is the speedup relative to the ideal?
- Is the speedup linear, super-linear, or worse?
- What about the efficiency?
- Is there a “plateau” in the speed?
- What is the best number of threads to use?

Monitoring the parallel program

- Use top
- How many %CPU is it using as you vary the number of threads?

Remember to quit the session

- After finishing the interactive session started with qlogin:

```
exit
```