

Concurrent Programming I

HPPS

David Marchant

Based on slides by:

Troels Henriksen, Randal E. Bryant and David R. O'Hallaron

Concurrent Programming

- So far we've talked about threads and processes each running a complete programme
- Each programme has been sequential in nature
- We can spread processing over several threads or processes, each part of the same programme
- Twice the resources means twice the speed! (Nope)

The Basic Idea

- We can break our processing over several workers
- Many hands make light work

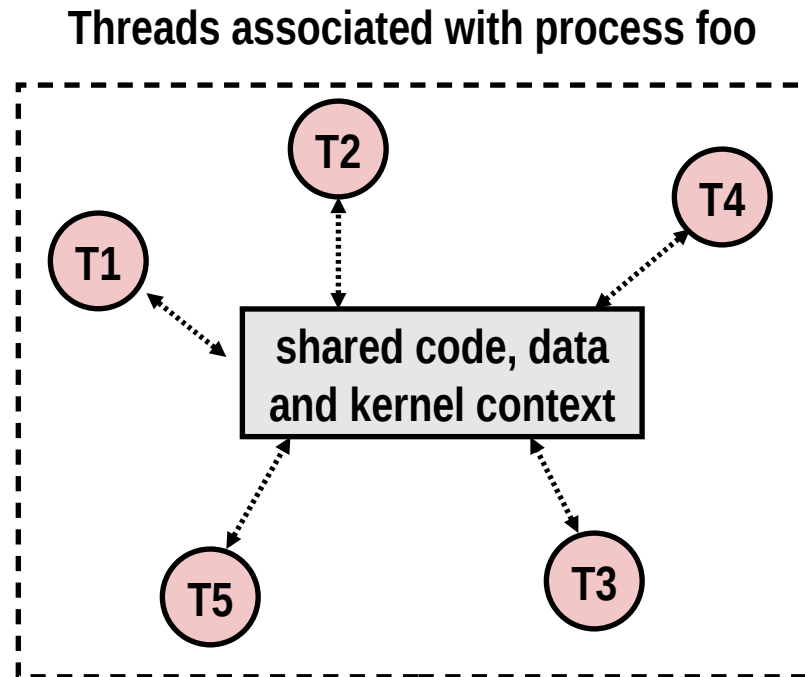
Single process
Runs for 10 mins



Two processes
Each run for 5 mins

Logical View of Threads

- Threads associated with process form a pool of peers
- Each thread can be processed in parallel

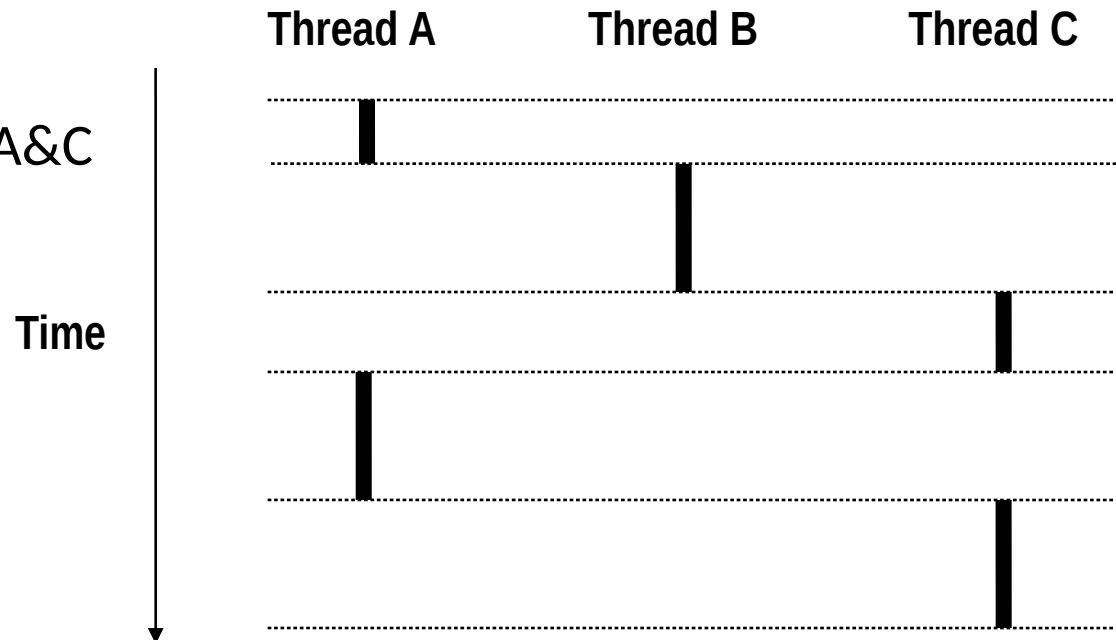


Concurrent Threads

- Two threads are *concurrent* if their flows overlap in time
- Otherwise, they are sequential

- **Examples:**

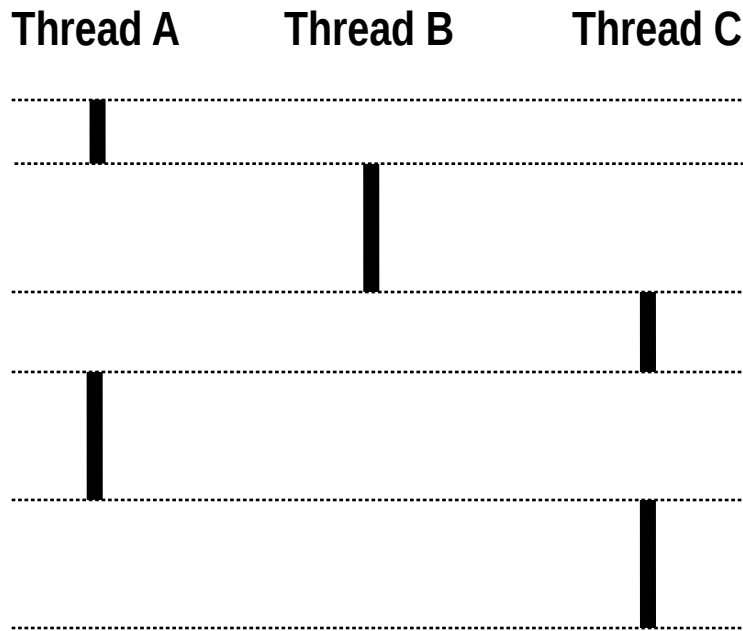
- Concurrent: A & B, A&C
- Sequential: B & C



Concurrent and Parallel Threads

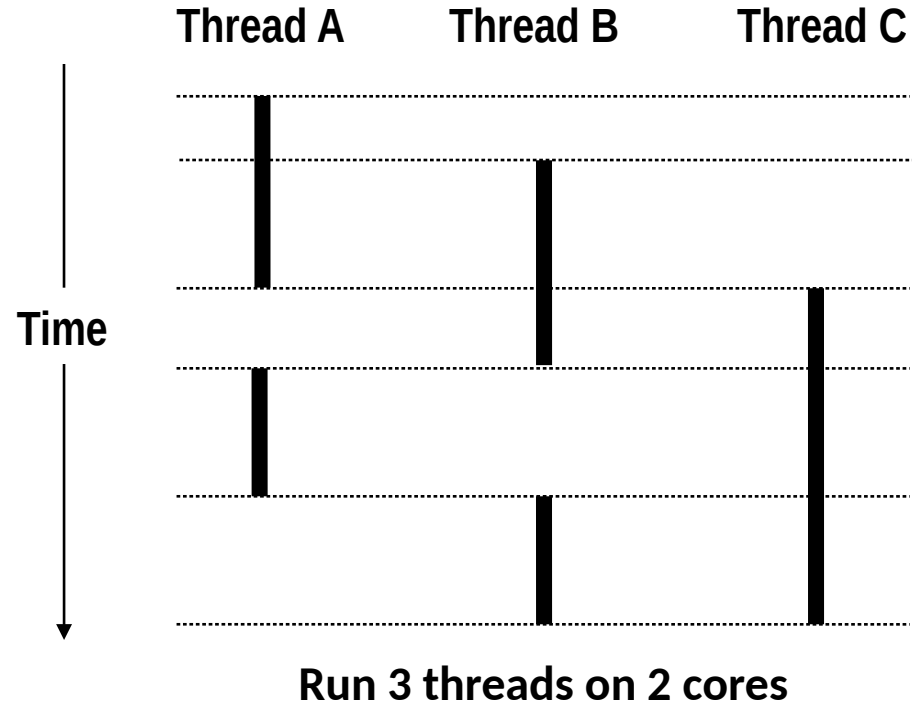
■ Single Core Processor

- Simulate parallelism by time slicing



■ Multi-Core Processor

- Can have true parallelism



Parallel Systems are the default

- For many years now multi-core systems have been standard
- If we paid for 8 cores, we want to use them!
- ...and not just on running background processes with slightly fewer context switches, we want a single programme to use all of the hardware

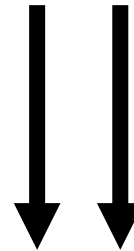
Parallelism is about speed.

- How do we quantify whether parallelisation made a program faster, and by how much?
- How can we estimate the potential benefit of parallelising a program?
- Are there limits to the potential gains of parallelism?
So far we've talked about threads and processes each running a complete programme

Latency

- Often called runtime
- How long it takes for a program to run on some workload.
- Both difficult and easy to measure
- *Wall time* is the time it takes in the real world.
- *CPU time* is the total amount of time spent executing code counting all processors.

Single process
Runs for 10 mins



Two processes
Each run for 5 mins

Here the CPU time is the same,
But the wall time has halved

Latency

- Suppose we have two programmes, P_1 and P_2 that each **perform the same task**, and take T_1 and T_2 respectively:

$$\text{Latency} = \frac{T_1}{T_2}$$

- Speedup greater than one means P_2 is faster than P_1 , else it is slower.
- P_1 and P_2 must solve the same problem for their latencies to be comparable.
- HINT: This is a good metric to include in any report on your own programmes

Throughput

- Sometimes latency is inappropriate—what is the running time of a potentially-infinite web server?
- Can measure latency for individual requests, but is that really useful?

$$\text{Throughput} = \frac{\text{Workload}}{\text{Time}}$$

- E.g. web requests served per second, or number of bytes processed per clock cycle.
- Using throughput we can compare programs that have different workloads.

Speedup in Throughput

- Suppose we have two programmes, P_1 and P_2 that each **perform the same task**, and have a throughput of Q_1 and Q_2 respectively:

$$\text{Throughput Speedup} = \frac{Q_1}{Q_2}$$

- Speedup greater than one means P_2 is faster than P_1 , else it is slower.
- P_1 and P_2 must solve the same problem for their throughputs to be comparable.
- HINT: This is another good metric to include in any report on your own programmes

Scalability

- Basically how well our system improves (or not) as we make it bigger

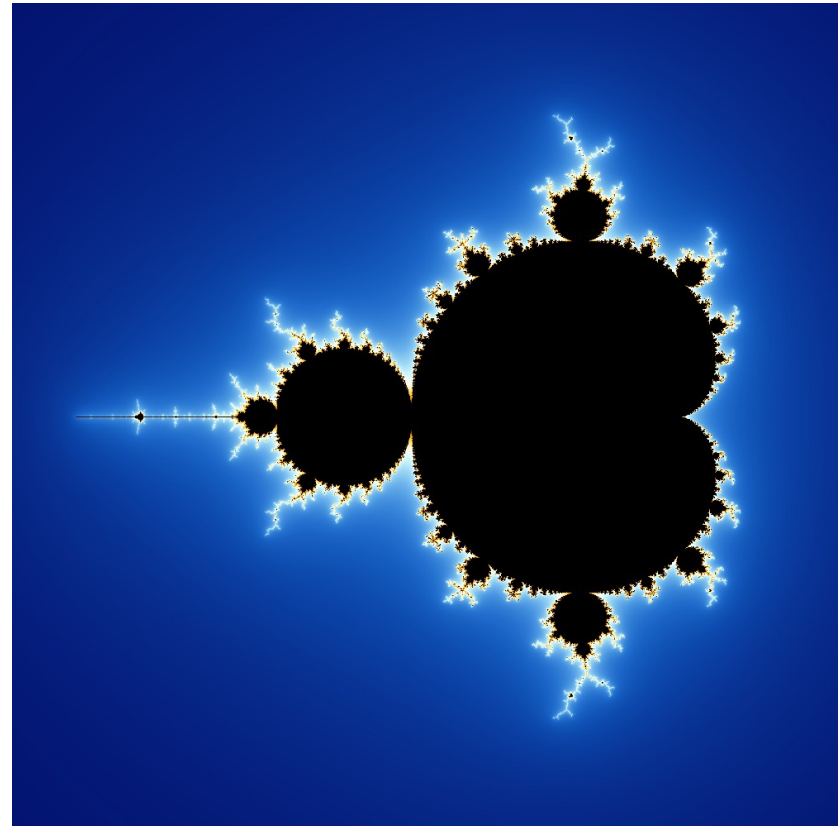
How the system improves in its capacity (runtime or throughput) as we add more resources, such as more processors

How the system throughput changes as we add more workload

- Both are similar, but we are altering different parameters in our system

Scalability Example

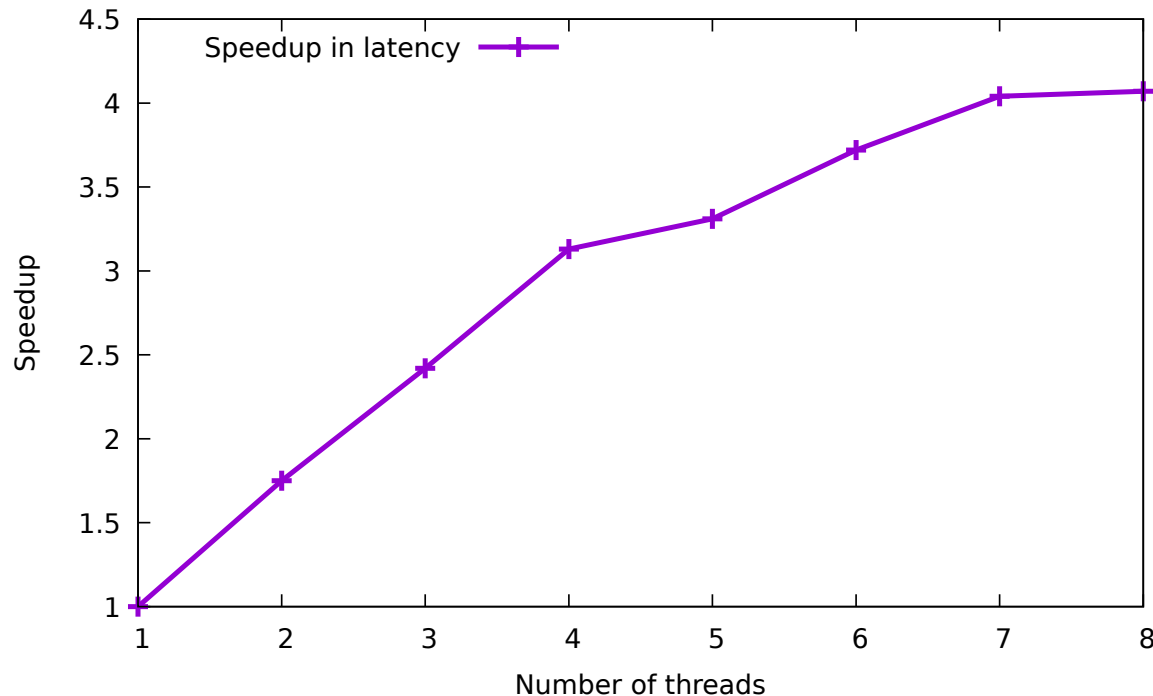
- Rendering Mandelbrot fractals
- Each pixel can be computed independently
- The image size corresponds to the workload
- Regarding parallelism, there are two sorts of scalability that we are interested in.



Strong Scaling

- How the runtime varies with the number of processors for a fixed problem size.

Speedup graph for rendering a 10^4 pixel Mandelbrot fractal as we use more threads:

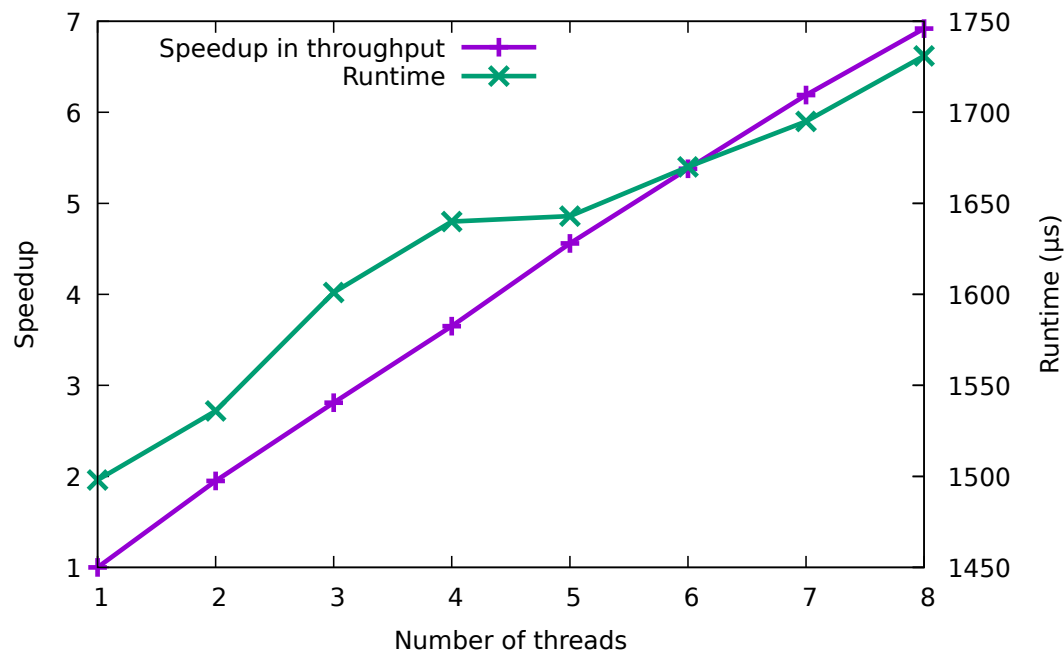


- This shows sub-linear strong scalability—8 threads barely gets us $4 \times$ speedup

Weak Scaling

- How the runtime varies with the number of processors for a fixed problem size relative to the number of processors.

Speedup graph for rendering a Mandelbrot fractal with 10^4 pixels per thread:



- Pretty decent weak scalability! By far more common than strong scalability.

How do we estimate the potential benefits of parallelising a program, without just implementing it and measuring how well it goes?

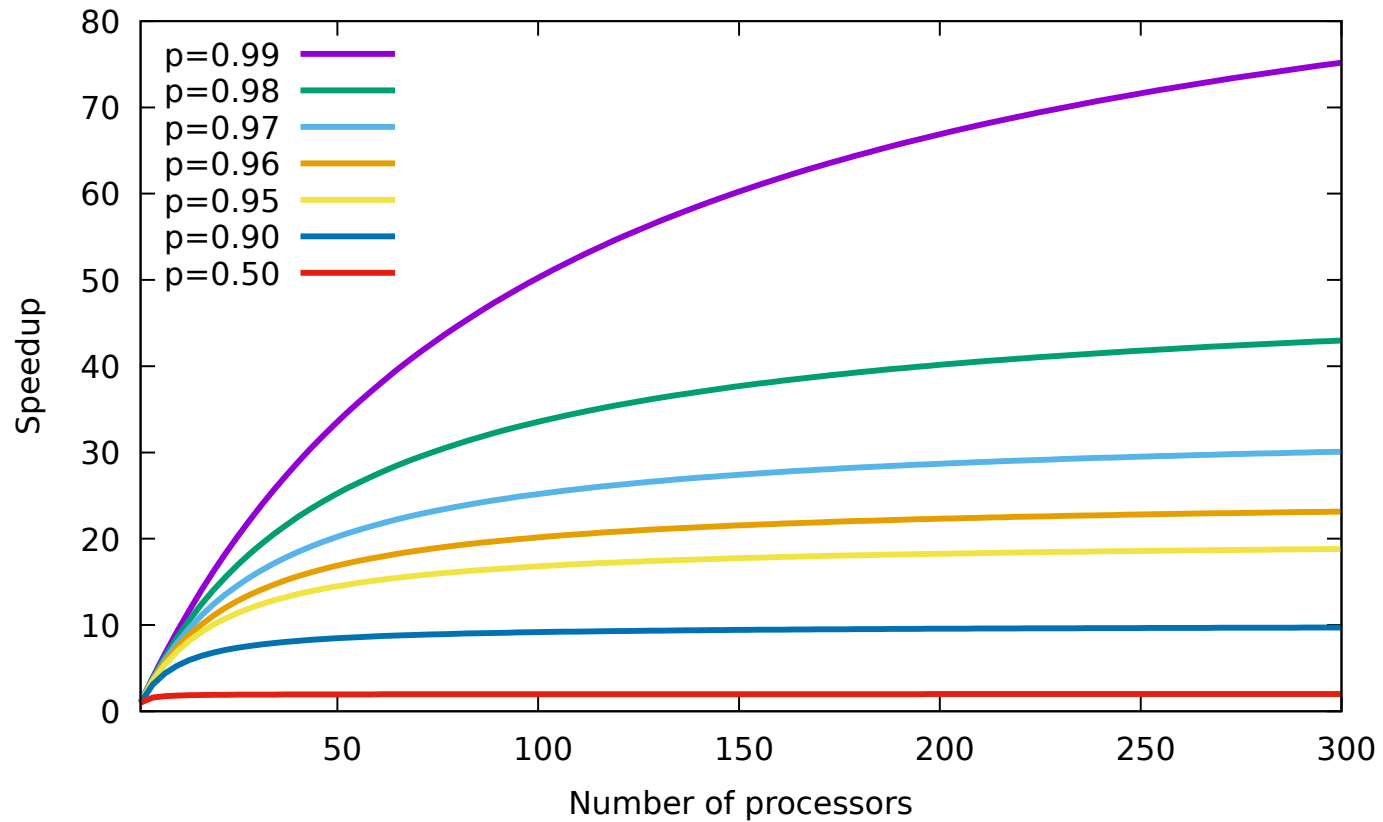
Amdahl's Law

- If p is the proportion of execution time that benefits from parallelisation, then $S(N)$ is maximum theoretical speedup achievable by execution on N processors, and is given by:

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}}$$

- Why is can p not equal 1?
- Reading, writing and other fundamentally sequential parts mean p will never be 1 in practice

Amdahl's Law



Amdahl's Law

- Amdahl's Law predicts strong scalability.
- For a fixed problem size, there will always come a point of diminishing returns.
- Makes parallelisation seem pointless...

But...

- You have a simulation that runs in 1 hour.
- Then you get a computer that is ten times faster.
- Usually you don't decide to run the simulation in six minutes, instead you make a simulation that is ten times as precise and still runs in 1 hour.
- Consider weather forecasting...
- Bigger machines let us solve bigger problems in the same time!

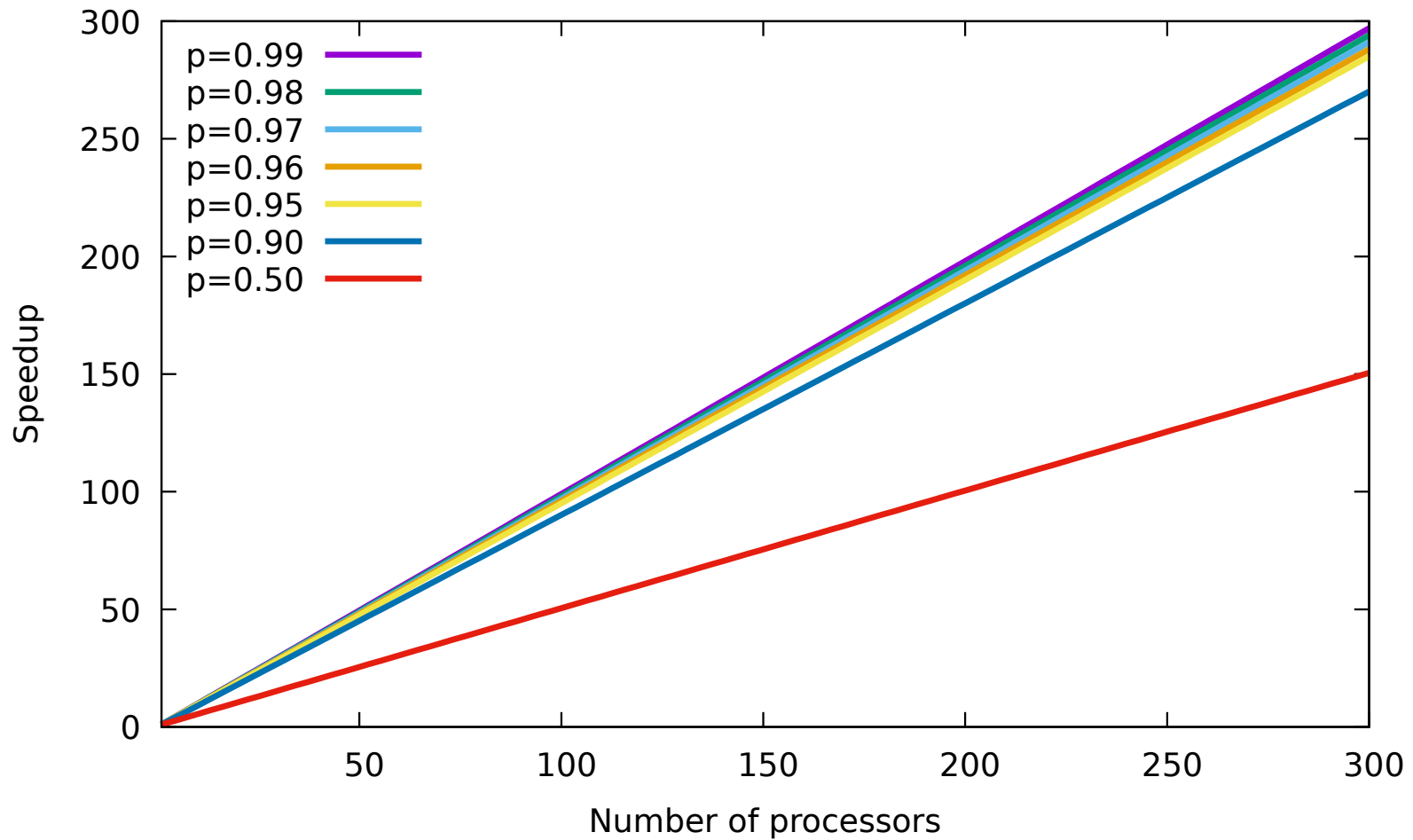
Gustadson's Law

- If s is the proportion of execution time that must be sequential, then $S(N)$ is maximum theoretical speedup achievable by execution on N processors, and is given by:

$$S(N) = N + (1-N) \times s$$

- Assumes that parallel workload increases just as fast as number of processors.
- Predicts weak scalability.
- In practice often more relevant than Amdahl's Law.

Gustadson's Law



The Fine Print

- Both Amdahl's and Gustafson's Laws are idealised abstractions and ignore important real-world concerns:
 - Locality.
 - Communication.
 - Synchronisation.

- But they are still a good theoretical framework for estimating the value of parallelising some program.

How do we start throwing threads at our problems?

Parallel Loops

- For scientific computing, we are mostly concerned with parallelising straightforward loops such as this matrix multiplication code

```
void matmul(int n , const double *x ,  
            const double *y , double *out ) {  
    for ( int i = 0 ; i < n ; i++) {  
        for ( int j = 0 ; j < n ; j++) {  
            out [ i *n+j ] = 0 ;  
            for ( int l = 0 ; l < n ; l++) {  
                out [ i *n+j ] += x [ i *n+j ] * y [ i *n+j ] ;  
            }  
        }  
    }  
}
```

- Iterations of the outer two loops are independent.
- Can be computed by different threads.

Simplest OpenMP Example

```
#pragma omp parallel for  
for (int i = 0 ; i < n ; i++) {  
    A[i] = A[i] * 2;  
}
```

- *Directives* used to indicate how run C program in parallel.
- *Clauses* (covered later) can be used to customise the behaviour.
- Semantics are the *sequential elision*—how the program would behave if we ignored the directives.
- We are only scratching the surface of OpenMP in this course!

Simplest OpenMP Example

```
print ("Program Starts\n");  
n = 1000;
```

```
#pragma omp parallel for  
for (int i=0; i<n; i++) {  
    A[i] = B[i] + C[i];  
}
```

```
m = 500;
```

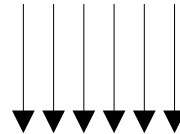
```
#pragma omp parallel for  
for (int j=0; j<m; j++) {  
    p[j] = q[j] - r[j];  
}
```

```
print ("Program Ends\n");  
exit(0),
```

Sequential



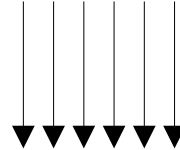
Parallel



Sequential



Parallel



Sequential



- Program starts sequential.
- Parallel regions split across multiple threads.
- Parallel region ends when all threads done.
- Worker threads kept running in background.

Compilation

```
#include <stdio.h>
#include <stdlib.h>

int main (void) {
    int n = 1000000000;
    int *arr = malloc (n*sizeof(int)) ;

    #pragma omp parallel for
    for (int i=0; i<n; i++) {
        arr[ i ] = i ;
    }

    free(arr);
}
```

```
$ gcc -o openmp-example openmp-example.c -fopenmp
```

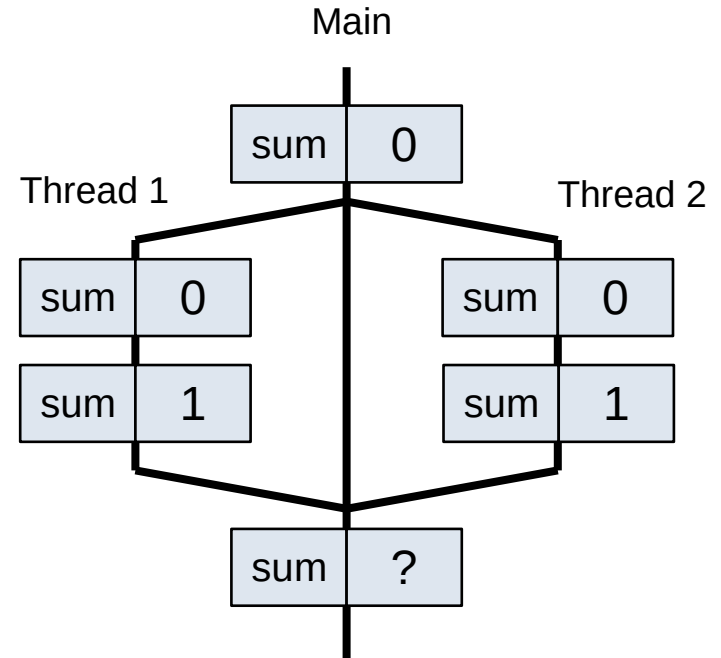
Controlling the number of threads

```
$ time OMP_NUM_THREADS=1 ./openmp-example
real    0m0.124s    Wall time
user    0m0.034s    CPU time (user)
sys     0m0.090s    CPU time (system)
$ time OMP_NUM_THREADS=2 ./openmp-example
real    0m0.076s    Wall time
user    0m0.033s    CPU time (user)
sys     0m0.104s    CPU time (system)
$ time OMP_NUM_THREADS=3 ./openmp-example
real    0m0.054s    Wall time
user    0m0.039s    CPU time (user)
sys     0m0.133s    CPU time (system)
$ time OMP_NUM_THREADS=4 ./openmp-example
real    0m0.046s    Wall time
user    0m0.054s    CPU time (user)
sys     0m0.184s    CPU time (system)
```

Memory model

- Variables declared inside a loop iteration are private.
- Variables declared outside are shared.
- Be extremely careful when modifying shared variables—OpenMP will not protect you!
- This is known as a race condition, multiple threads are reading and writing to the same location. Who gets there first? We don't know!
- We will cover this in more depth next week

```
Double sum=0;  
#pragma omp parallel for  
for (int i = 0 ; i < n ; i++) {  
    sum += A[i];  
}
```



Reductions

- Instead of doing a summation sequentially

$$(((((((x_0 + x_1) + x_2) + x_3) + x_4) + x_5) + x_6) + x_7)$$

- we can do it like:

$$(x_0 + x_1 + x_2 + x_3) + (x_4 + x_5 + x_6 + x_7)$$

- and have one compute the left part, and a second one compute the right, combining their results at the end.

Is this valid?

Reductions

- Instead of doing a summation sequentially

$$(((((((x_0 \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4) \oplus x_5) \oplus x_6) \oplus x_7)$$

- we can do it like:

$$(x_0 \oplus x_1 \oplus x_2 \oplus x_3) \oplus (x_4 \oplus x_5 \oplus x_6 \oplus x_7)$$

- and have one compute the left part, and a second one compute the right, combining their results at the end.

What about now?

Reductions

- A binary operator $\oplus : \alpha \rightarrow \alpha \rightarrow \alpha$ is said to be *commutative* if

$$x \oplus y = y \oplus x$$

- It is *associative* if

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

- We can do a parallel “sum” with any associative operator, called a reduction.
- For convenience, we also tend to require a neutral element 0_{\oplus} :

$$x \oplus 0_{\oplus} = 0_{\oplus} \oplus x = x$$

Reduction example

```
double dotprod (int n, double *x, double *y) {  
    double sum = 0 ;  
    #pragma omp parallel for reduction (+: sum)  
    for (int i=0; i<n; i++) {  
        sum += x[i] * y[i] ;  
    }  
    return sum ;  
}
```

- Must initialise accumulator variable to neutral element.
- Must explicitly tell OpenMP the combining operator (+, *, -, &&, ||, &, |, ^, max, or min).

Parallelising Matrix Multiplication

```
void matmul_seq(int n , const double *x ,  
                const double *y, double *out) {  
    for (int i=0; i<n; i++) {  
        for (int j=0; j<n; j++) {  
            double acc = 0;  
            for (int l=0; l<n; l++) {  
                acc += x[i*n+l] * y[j*n+l] ;  
            }  
            out[i*n+j] = acc;  
        }  
    }  
}
```

- Runtime for n=1000: 2.69s.
- Three nested loops.
- Which do we parallelise, and how?

Parallelising Matrix Multiplication

```
void matmul_seq(int n , const double *x ,
               const double *y, double *out) {
    #pragma omp parallel for
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            double acc = 0;
            for (int l=0; l<n; l++) {
                acc += x[i*n+l] * y[j*n+l] ;
            }
            out[i*n+j] = acc;
        }
    }
}
```

- Runtime: 0.80s (was. 2.69s seq) — pretty good for adding one line of code!
- But this only parallelises across the rows of the result matrix.

Parallelising Matrix Multiplication

```
void matmul_seq(int n , const double *x ,
               const double *y, double *out) {
    #pragma omp parallel for collapse(2)
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            double acc = 0;
            for (int l=0; l<n; l++) {
                acc += x[i*n+l] * y[j*n+l] ;
            }
            out[i*n+j] = acc;
        }
    }
}
```

- collapse(2) clause combines loops to one parallel $n \times n$ iteration loop,
- Runtime: 0.93s (was 0.80s without collapse) — not much impact here.
- Would matter more if we had fewer iterations in outer loop.

Parallelising Matrix Multiplication

```
void matmul_seq(int n , const double *x ,
                const double *y, double *out) {
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            double acc = 0;
            #pragma omp parallel for reduction(+:acc)
            for (int l=0; l<n; l++) {
                acc += x[i*n+l] * y[j*n+l] ;
            }
            out[i*n+j] = acc;
        }
    }
}
```

- Runtime: 2.48s (vs. 2.69s sequential)—this sucks.
- Almost always better to parallelise outermost loops.

Parallelising Matrix Multiplication

```
void matmul_seq(int n , const double *x ,  
                const double *y, double *out) {  
    #pragma omp parallel for  
    for (int j=0; j<n; j++) {  
        for (int i=0; i<n; i++) {  
            double acc = 0;  
            for (int l=0; l<n; l++) {  
                acc += x[i*n+l] * y[j*n+l] ;  
            }  
            out[i*n+j] = acc;  
        }  
    }  
}
```

- Runtime: 0.80s (was. 0.80 first ordering)
- No real effect here as both loop are the same, but can exploit locality for some small but easy speedup

Scheduling Clauses

- By default, OpenMP splits the iterations of a parallel loop evenly among the threads (static scheduling). This is not always optimal.

```
int fib (int n) {  
    if (n <= 1) {  
        return 1;  
    } else {  
        return fib(n-1) + fib(n-2);  
    }  
}
```

- Consider a loop that computes fib(i) for each $i < n$.
- Since time to compute fib(i) is over twice that of fib(i-1), the threads with the early iterations finish much faster than the threads with the later iterations.

```
#pragma omp parallel for schedule(static)  
for (int i=0; i<n; i++) {  
    fibs[i] = fib(i);  
}
```


Dynamic Scheduling

- Dynamic scheduling assigns each thread an iteration, and is given more iterations when it finishes.
- No idle threads (as long as there are unclaimed iterations to run).

```
#pragma omp parallel for schedule(dynamic)  
for (int i=0; i<n; i++) {  
    fibs[i] = fib(i);  
}
```

- On my machine static ran in 5.2s, where dynamic ran in 2.27s—much better!
- By default, assigns single iterations at a time, which is slow for very large loops with quick iterations.
- Use `schedule(dynamic,K)` to schedule K iterations at a time to threads.

Summary

- Parallelism is about speedup.
- Describe performance differences with speedup.
 - Latency for programs that compute the same thing.
 - Throughput if not.
- Strong scaling is solving the same problem faster.
- Weak scaling is solving a bigger problem in the same time.
- OpenMP is a simple language extension for parallelising loops in C programs.
- Use `#pragma omp parallel for` to parallelise a for-loop.
- Generally parallelising the outer most loop works best