The University of Manchester

COMP35112 Chip Multiprocessors

Parallel Programming Using Shared Memory

Pierre Olivier

Threads

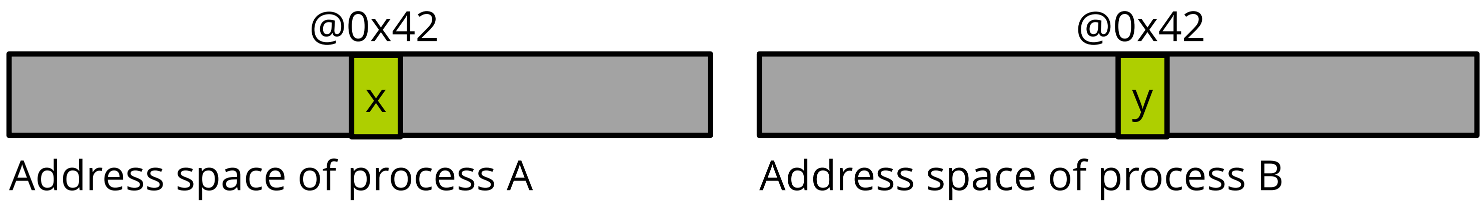
Processes

- A program executing on the CPU runs as a **process**
- With virtual memory, the process gets the illusion it has access to 100% of the memory
  - **Address space**

Processes

- A program executing on the CPU runs as a **process**
- With virtual memory, the process gets the illusion it has access to 100% of the memory
  - **Address space**

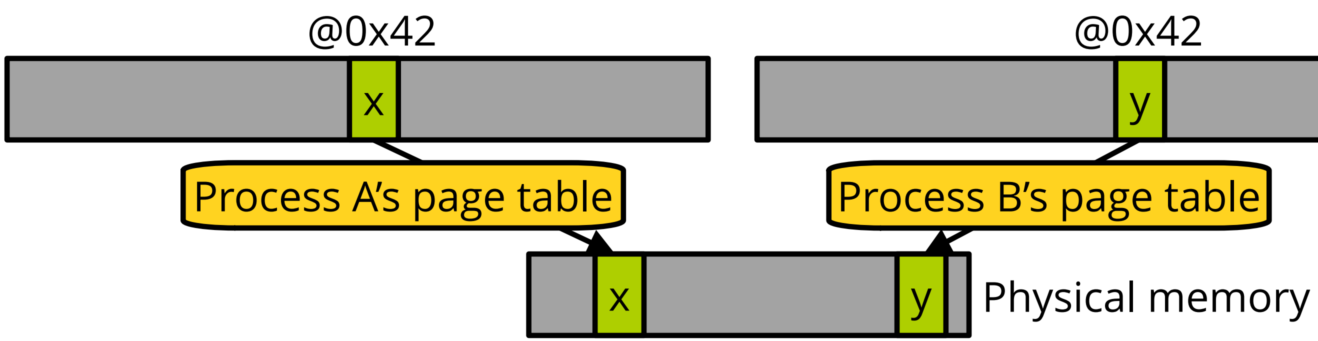
- Two programs run in 2 different processes, i.e. 2 different address spaces



Processes

- A program executing on the CPU runs as a **process**
- With virtual memory, the process gets the illusion it has access to 100% of the memory
  - **Address space**

- Two programs run in 2 different processes, i.e. 2 different address spaces

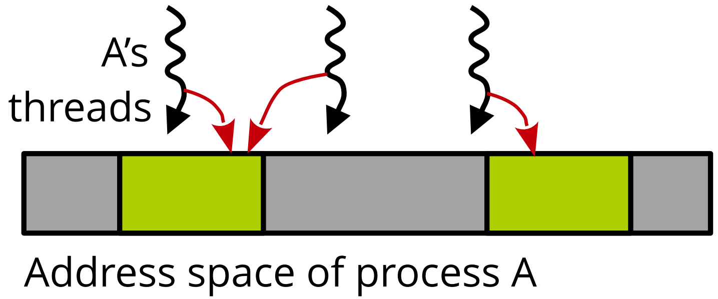


Threads

- A thread is a sequence of instructions executed on a CPU core
- A process can consist of one or more threads
- Each process in the system has an address space
- **All threads in the same process share the same address space**

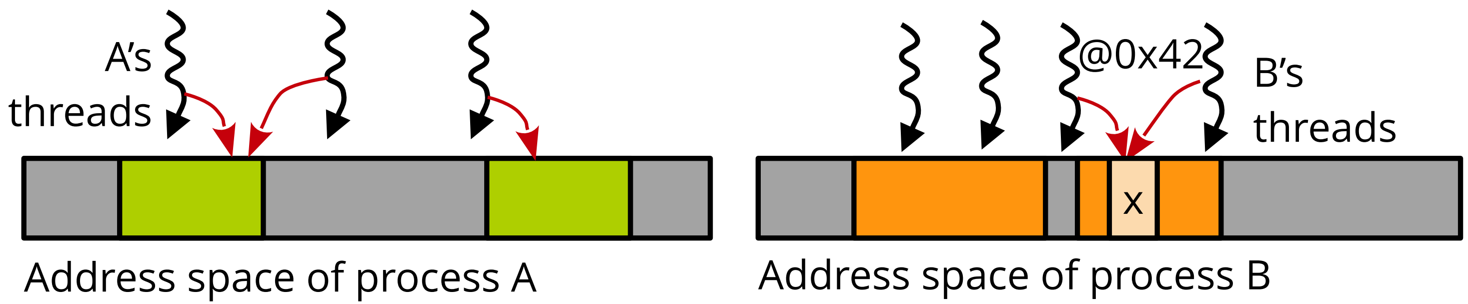
Threads

- A thread is a sequence of instructions executed on a CPU core
- A process can consist of one or more threads
- Each process in the system has an address space
- **All threads in the same process share the same address space**



Threads

- A thread is a sequence of instructions executed on a CPU core
- A process can consist of one or more threads
- Each process in the system has an address space
- **All threads in the same process share the same address space**



Threads

- A thread is a sequence of instructions executed on a CPU core
- A process can consist of one or more threads
- Each process in the system has an address space
- **All threads in the same process share the same address space**

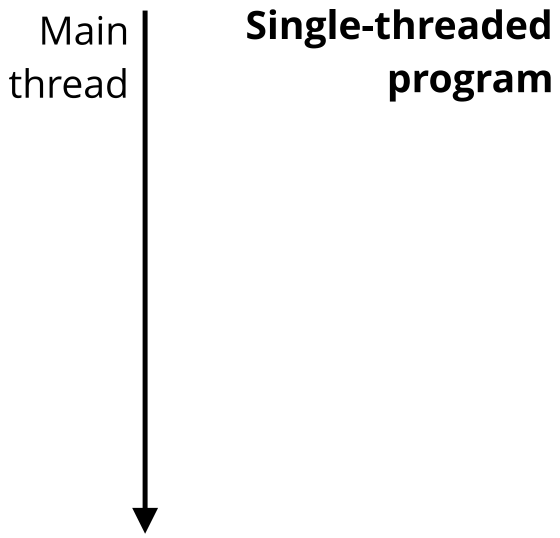
Threads communicate using shared memory

Threads

- Can program with threads in:
  - C/C++/Fortran – using the POSIX threads (Pthread) library
  - Java
  - Many other languages: Python, C#, Haskell, Rust, etc.

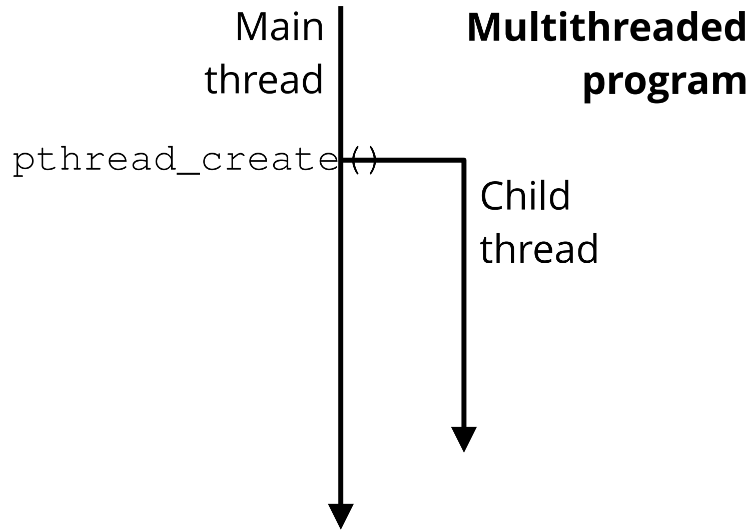
Threads in C/C++ with Pthread

- **Pthread** stands for the **POSIX thread library**
- Expose a C/C++ API to create and manage multiple threads within a process
- Standard program: OS creates automatically a single thread



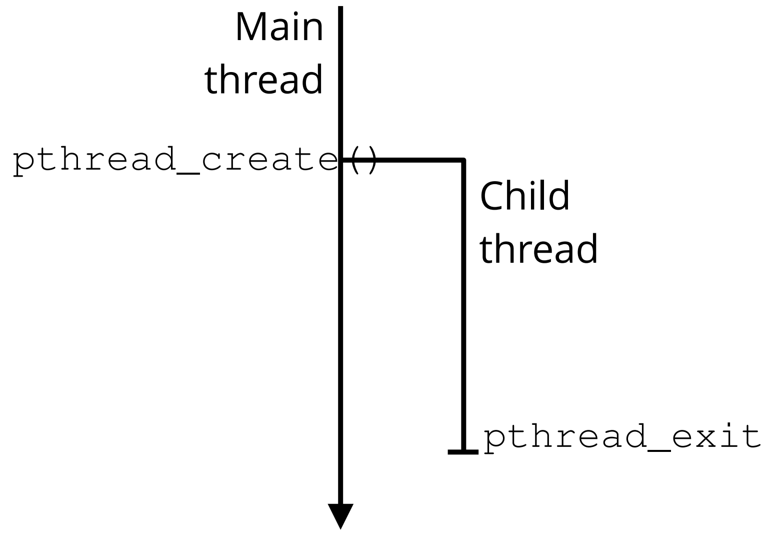
Threads in C/C++ with Pthread

- Use `pthread_create()` to create and launch a thread
  - Indicate as parameters:
    - Which function the thread should run
    - Optionally what should be passed as parameter to this function



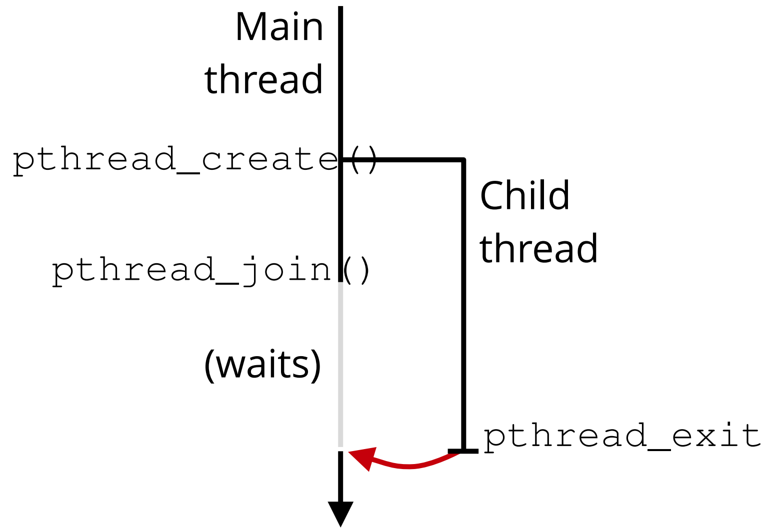
Threads in C/C++ with Pthread

- Use `pthread_create()` to create and launch a thread
  - Indicate as parameters:
    - Which function the thread should run
    - Optionally what should be passed as parameter to this function
- `pthread_exit()` to have the calling thread exit



Threads in C/C++ with Pthread

- Pthread -- POSIX thread library
- Use `pthread_create()` to create and launch a thread
    - Indicate as parameters:
        - Which function the thread should run
        - Optionally what should be passed as parameter to this function
- `pthread_exit()` to have the calling thread exit
- `pthread_join()` to wait for another thread to finish



Threads in C/C++ with Pthread

- A good chunk of this course, including labs 1 and 2, will focus on shared memory programming in C/C++ with pthreads
- `man pthread_*` and Google “pthreads” for lots of documentation
  - In particular see the Oracle Multithreaded Programming Guide: https://bit.ly/3FGt3k2

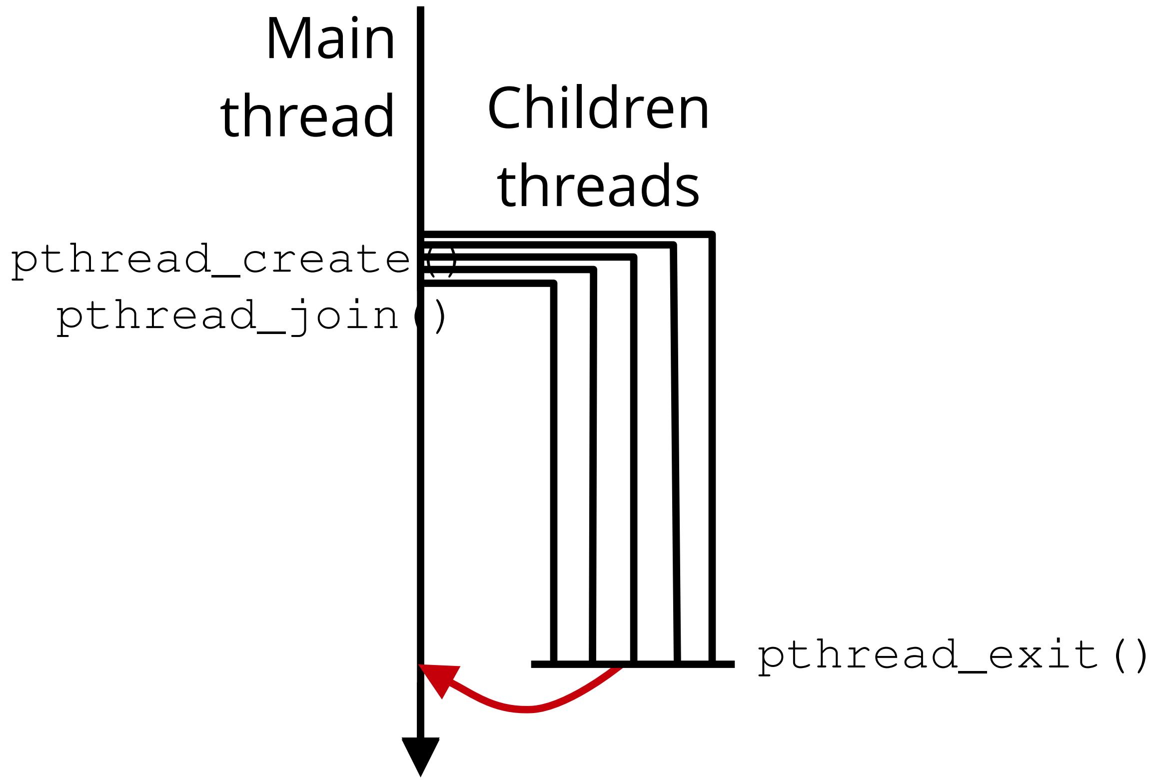
Threads in C/C++ with Pthread

```c
// Compile with:
// gcc pthread.c -o pthread -lpthread

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

#define NOWORKERS 5

// Function executed by all threads
void *thread_fn(void *arg) {
    int id = (int)(long)arg;

    printf("Thread %d running\n", id);

    pthread_exit(NULL);  // exit

    // never reached
}
```

```c
int main(void) {
  // Each thread is controlled through a
  // pthread_t data structure
  pthread_t workers[NOWORKERS];

  // Create and launch the threads
  for(int i=0; i<NOWORKERS; i++)
    if(pthread_create(&workers[i], NULL,
        thread_fn, (void *)(long)i)) {
      perror("pthread_create");
      return -1;
    }

  // Wait for threads to finish
  for (int i = 0; i < NOWORKERS; i++)
    if(pthread_join(workers[i], NULL)) {
      perror("pthread_join");
      return -1;
  }

  printf("All done\n");
}
```

03-shared-memory-programming/pthread.c

Threads in C/C++ with Pthread



Threads in Java

- Two ways of defining a Thread
  - Class inherits from `java.lang.Thread`
  - Class implements `java.lang.Runnable`
    - Lets you inherit from something else than `Thread`

Threads in Java

- Two ways of defining a Thread
  - Class inherits from `java.lang.Thread`
  - Class implements `java.lang.Runnable`
    - Lets you inherit from something else than `Thread`
- In both cases, `run()` method defines what the thread does when it starts running
- `Thread.start()` gets it going
- Can use `Thread.join()` to wait for it to complete

Threads in Java

```java
class MyThread extends Thread {
    int id;
    MyThread(int id) { this.id = id; }
    public void run() { System.out.println("Thread " + id + " running"); }
}

class Demo {
    public static void main(String[] args) {
        int NOWORKERS = 5;
        MyThread[] threads = new MyThread[NOWORKERS];

        for (int i = 0; i < NOWORKERS; i++)
            threads[i] = new MyThread(i);
        for (int i = 0; i < NOWORKERS; i++)
            threads[i].start();

        for (int i = 0; i < NOWORKERS; i++)
            try {
                threads[i].join();
            } catch (InterruptedException e) { /* do nothing */ }
        System.out.println("All done");
    }
}
// compile and launch with:
//javac java-thread.java && java Demo
```

03-shared-memory-programming/java-thread.java

Threads in Java

```
class MyRunnable implements Runnable {
    int id;
    MyRunnable(int id) { this.id = id; }
    public void run() { System.out.println("Thread " + id + " running"); }
}

class Demo {
    public static void main(String[] args) {
        int NOWORKERS = 5;
        Thread[] threads = new Thread[NOWORKERS];
        for (int i = 0; i < NOWORKERS; i++) {
            MyRunnable r = new MyRunnable(i);
            threads[i] = new Thread(r);
        }
        for (int i = 0; i < NOWORKERS; i++)
            threads[i].start();

        for (int i = 0; i < NOWORKERS; i++)
            try {
                threads[i].join();
            } catch (InterruptedException e) { /* do nothing */ }
        System.out.println("All done");
    }
}
// compile and launch with:
// javac java-runnable.java && java Demo
```

03-shared-memory-programming/java-runnable.java

Examples Output

For both Java and C examples:

```
Thread 1 running
Thread 0 running
Thread 2 running
Thread 4 running
Thread 3 running
All done
```

- **No control over the order of execution!**
  - The OS scheduler decides, it's nondeterministic

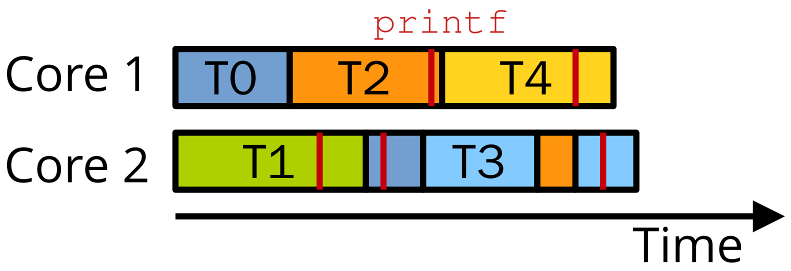
Examples Output

For both Java and C examples:

```
Thread 1 running
Thread 0 running
Thread 2 running
Thread 4 running
Thread 3 running
All done
```

- **No control over the order of execution!**
  - The OS scheduler decides, it's nondeterministic

A possible scheduling scenario:



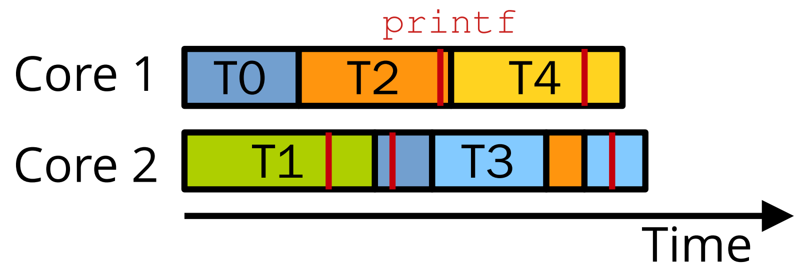
Examples Output

For both Java and C examples:

```
Thread 1 running
Thread 0 running
Thread 2 running
Thread 4 running
Thread 3 running
All done
```

- **No control over the order of execution!**
  - The OS scheduler decides, it's nondeterministic

A possible scheduling scenario:



Another one on 1 core:

Data Parallelism

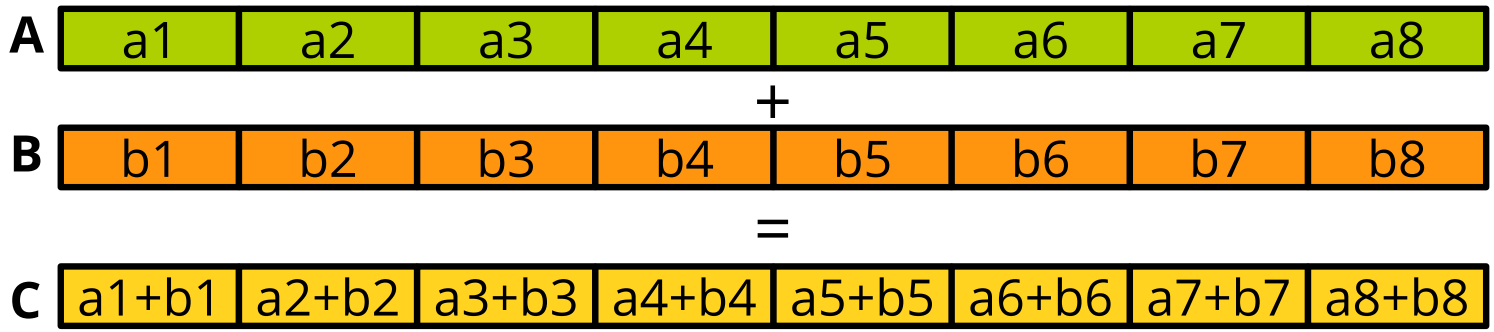
Dividing Work between Threads

Data Parallelism

- Simple form of parallelism commonly found in many applications
  - Common in computational science applications
- Divide computation into (nearly) equal sized chunks
- Works best when there are no data dependencies between chunks

Data Parallelism

- Simple form of parallelism commonly found in many applications
  - Common in computational science applications
- Divide computation into (nearly) equal sized chunks
- Works best when there are no data dependencies between chunks



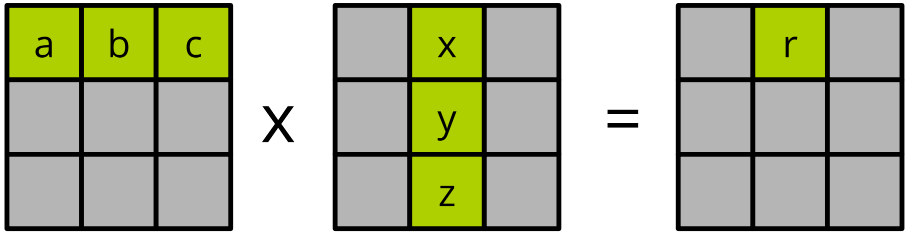
- Exploited in multithreading but also at the instruction level in vector/array (SIMD) processors: CPUs (SSE, AVX), and in GPGPUs

Data Parallelism Example

- Matrix multiply of n x n matrices is a good example

Data Parallelism Example

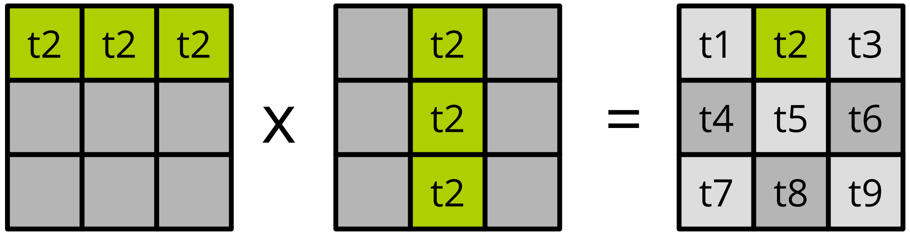
- Matrix multiply of n x n matrices is a good example



r = a * x + b * y + c * z

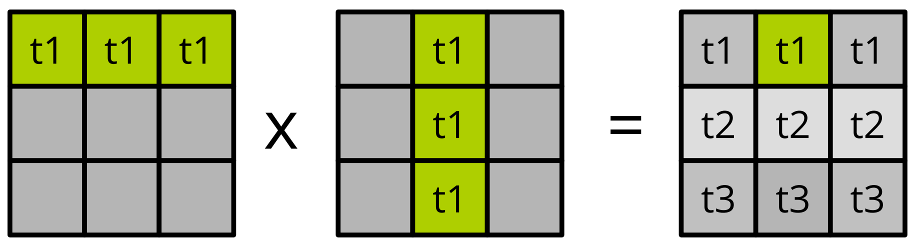
Data Parallelism Example

- Matrix multiply of n x n matrices is a good example
  - n^2 parallel threads (1 per result element)



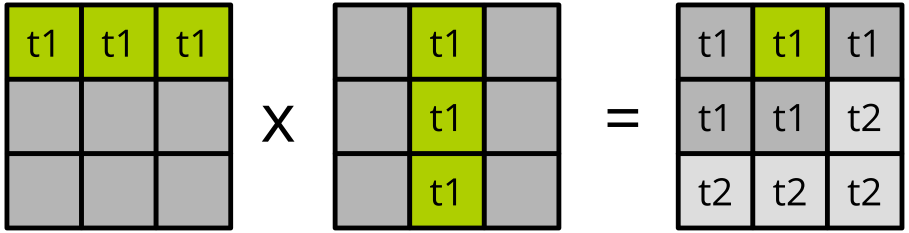
Data Parallelism Example

- Matrix multiply of n x n matrices is a good example
  - n^2 parallel threads (1 per result element)
  - n parallel threads (1 per row/column of result)



Data Parallelism Example

- Matrix multiply of n x n matrices is a good example
  - n^2 parallel threads (1 per result element)
  - n parallel threads (1 per row/column of result)
  - p parallel threads, each computing q rows/columns of the result, where $pq = n$



Data Parallelism Example

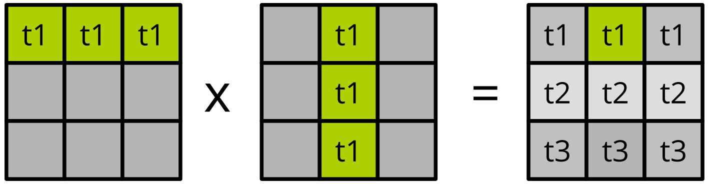
- Matrix multiply of n x n matrices is a good example
  - n^2 parallel threads (1 per result element)
  - n parallel threads (1 per row/column of result)
  - p parallel threads, each computing q rows/columns of the result, where $pq = n$
- Two important questions regarding the programmer's **effort**:
  - **What is the best strategy according to the situation?**
    - Does the programmer need to be an expert to make that choice?
  - **How to indicate the chosen strategy in the code?**
    - If we already have a sequential version of the program, how much code refactoring & new code implementation is needed?

Implicit vs. Explicit Parallelism

NxN Parallel Matrix Multiplication

```c
#define N 1000
int A[N][N]; int B[N][N]; int C[N][N];

int main(int argc, char **argv) {

  /* init matrices here */

  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++) {
      C[i][j] = 0;
      for(int k=0; k<N; k++)
          C[i][j] += A[i][k] * B[k][j];
    }

    return 0;
}
```

03-shared-memory-programming/matmult.c

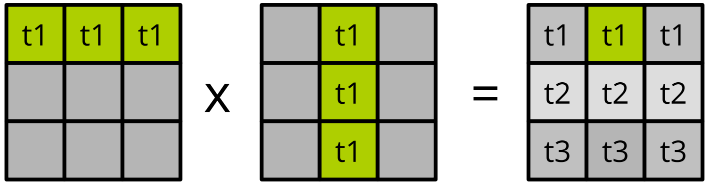


- Basic sequential code for matrix multiplication

NxN Parallel Matrix Multiplication

```c
#include <omp.h>
#define N 1000
int A[N][N]; int B[N][N]; int C[N][N];

int main(int argc, char **argv) {

  /* init matrices here */

#pragma omp parallel
  {
    /* First loop parralelised */
    for(int i=0; i<N; i++)
      for(int j=0; j<N; j++) {
        C[i][j] = 0;
        for(int k=0; k<N; k++)
          C[i][j] += A[i][k] * B[k][j];
      }
  }

    return 0;
}
```

[03-shared-memory-programming/openmp.c]()



- Automatic parallelisation using OpenMP
  - Very low programmer effort
  - More on OpenMP later in the course

Explicit vs. Implicit Parallelism

- **Explicit parallelism**
    - The programmer explicitly spells out what should be done in parallel/sequence
    - Code modifications needed if sequential program already available
    - Examples: using threads or other high level notations (e.g. OpenMP)

Explicit vs. Implicit Parallelism

- **Explicit parallelism**
  - The programmer explicitly spells out what should be done in parallel/sequence
  - Code modifications needed if sequential program already available
  - Examples: using threads or other high level notations (e.g. OpenMP)
- **Implicit parallelism**
  - No effort from the programmer, system works out parallelism by itself
  - No code modification over an already existing sequential program
  - Done for example by some languages able to make strong assumptions about data sharing (e.g. pure functions), or with ILP

Example Code for Implicit Parallelism

Some languages (e.g. Matlab, C++ and Java - via libraries, Fortran after f90) allow expressions on arrays:

```
A = B + C
```

with no side effects:

```
A = f(B) + g(C)
```

or even:

```
p = h(f(A), g(B))
```

Automatic Parallelisation

- In an ideal world, the compiler would **take an ordinary sequential program and derive the parallelism automatically**

Automatic Parallelisation

- In an ideal world, the compiler would **take an ordinary sequential program and derive the parallelism automatically**
    - Manufacturers of pre-multicore parallel machines invested considerably in such technology
    - Can do quite well if the programs are simple enough but **dependency analysis can be very hard**
    - Must be conservative
        - If you cannot be certain that parallel version computes **correct** result, can't parallelise

Example Problems for Parallelisation

- Can the compiler automatically parallelise the execution of these loops' iterations?
  - I.e. run all or some iterations in parallel

```
for (int i = 0 ; i < n-3 ; i++) {
  a[i] = a[i+3] + b[i] ;           // at iteration i, read dependency with index i+3
}

for (int i = 5 ; i < n ; i++) {
  a[i] += a[i-5] * 2 ;             // at iteration i, read dependency with index i-5
}

for (int i = 0 ; i < n ; i++) {
  a[i] = a[i + j] + 1 ;            // at iteration i, read dependency with index ???
}
```

Automatic Parallelisation

```
for (int i=0; i<n-3; i++)
  a[i] = a[i+3] + b[i];
```

- If we parallelise and iteration 3 runs before iteration 0 we break the program
- Positive offset: we read at each iteration what was in the array before the loop started
- **We never read a value computed by the loop itself**

i = 0	a[0] = a[3] + b[0]
i = 1	a[1] = a[4] + b[1]
i = 2	a[2] = a[5] + b[2]
i = 3	a[3] = a[6] + b[3]
i = 4	a[4] = a[7] + b[4]
	etc.

Automatic Parallelisation

```
for (int i=0; i<n-3; i++)
  a[i] = a[i+3] + b[i];
```

- If we parallelise and iteration 3 runs before iteration 0 we break the program
- Positive offset: we read at each iteration what was in the array before the loop started
- **We never read a value computed by the loop itself**

i = 0	a[0] = a[3] + b[0]
i = 1	a[1] = a[4] + b[1]
i = 2	a[2] = a[5] + b[2]
i = 3	a[3] = a[6] + b[3]
i = 4	a[4] = a[7] + b[4]
	etc.

- Can parallelise by making a new version of array `a`

```
parrallel_for(int i=0; i<n-3; i++)
    new_a[i] = a[i+3] + b[i];
a = new_a;
```

Automatic Parallelisation

```
for (int i = 5 ; i < n ; i++) {
      a[i] += a[i-5] * 2 ;
}
```

- Previous trick does not work here: **this time we read values computed by the loop itself**
- At each iteration i we read what was computed by the loop at iteration i-5

i = 5	a[5] = a[5] + a[0] *2
i = 6	a[6] = a[6] + a[1] *2
i = 7	a[7] = a[7] + a[2] *2
i = 8	a[8] = a[8] + a[3] *2
i = 9	a[9] = a[9] + a[4] *2
i = 10	a[10] = a[10] + a[5] *2
i = 11	a[11] = a[11] + a[6] *2
i = 12	a[12] = a[12] + a[7] *2
i = 13	a[13] = a[13] + a[8] *2
i = 14	a[14] = a[14] + a[9] *2
i = 15	a[15] = a[15] + a[10] *2
	etc.

Automatic Parallelisation

```
for (int i = 5 ; i < n ; i++) {
      a[i] += a[i-5] * 2 ;
}
```

- Previous trick does not work here: **this time we read values computed by the loop itself**
- At each iteration i we read what was computed by the loop at iteration i-5
- **Solution: limit parallelism to 5**

i = 5	a[5] = a[5] + a[0] *2
i = 6	a[6] = a[6] + a[1] *2
i = 7	a[7] = a[7] + a[2] *2
i = 8	a[8] = a[8] + a[3] *2
i = 9	a[9] = a[9] + a[4] *2
i = 10	a[10] = a[10] + a[5] *2
i = 11	a[11] = a[11] + a[6] *2
i = 12	a[12] = a[12] + a[7] *2
i = 13	a[13] = a[13] + a[8] *2
i = 14	a[14] = a[14] + a[9] *2
i = 15	a[15] = a[15] + a[10] *2
	etc.

Shared Memory

- For this lecture we assumed that threads **share memory**
  - I.e. they all have access to a common address space
  - Multiple threads can read and write in the same memory location (variable, buffer, etc.) through
    - Global variable
    - Pointers and references
- No shared memory?
  - Need to communicate via **message passing**
  - Close to a distributed system
  - We'll talk briefly about MPI (Message Passing Interface) later in the course unit

Summary and Next Lecture

- **In shared memory systems, a parallel program can use threads**
  - Threads communicate implicitly by reading and writing to a common address space
- **A simple form of parallelism: data parallelism**
  - Apply similar operations on chunks of a data set
  - Efficient when there is no data dependency
- **Automatic parallelisation can be limited by such dependencies**
- Shared memory is a practical form of implicit communication
  - It's great that multicore today share memory right?
  - Next lecture will discuss why this isn't as simple as it sounds!