Parallel design patterns ARCHER course

Vectorisation and active messaging

Reusing this material

Finding Concurrency

- Task Decomposition, Data Decomposition, Group Tasks, Order Tasks, …

Algorithm Structure

- Tasks Parallelism, Divide and Conquer, Geometric Decomposition, Recursive Data, …

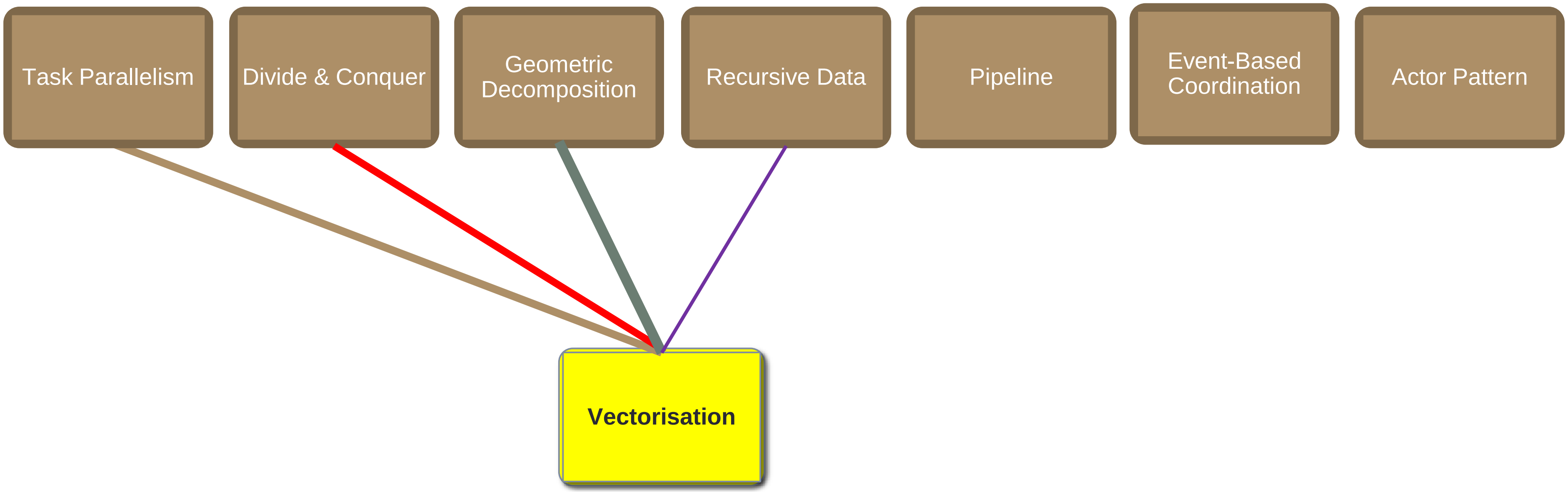
Supporting Structures

- SPMD, Master/Worker, Loop Parallelism, Fork/Join, …

Implementation Mechanisms

- UE Management, Synchronisation, Communication, …

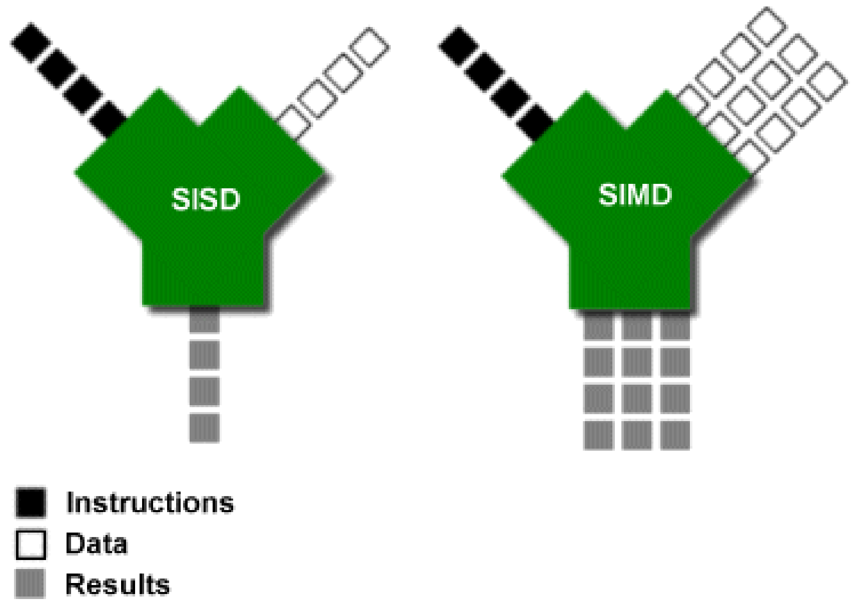
Vectorisation: The Problem



- Vectorisation is an *Implementation Strategy*
- The Problem: Given a program whose run time is dominated by a set of calculations, how can this be translated into a parallel program?
- Also known as SIMD

Single Instruction Multiple Data

- Single stream of instructions operating on multiple data streams

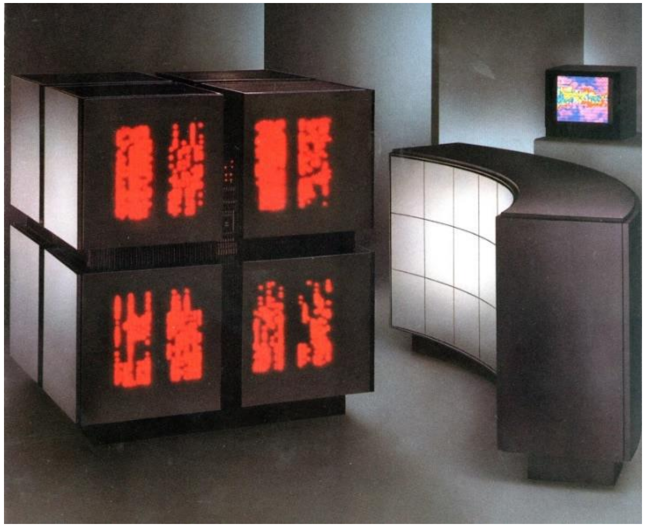


- The problem is typically defined in terms of arrays that can be updated concurrently using the same instructions
- Create a single stream of instructions
  - Can have a mask to allow for some selection based on data
- Can work well when your problem is truly data parallel

epcc

Trying to force SIMD through code

```
int inputNumbers[1000];

int i,finalSum;

finalSum=0;
for (i=0;i<=999;i++) {
  finalSum+=inputNumbers[i];
}
```

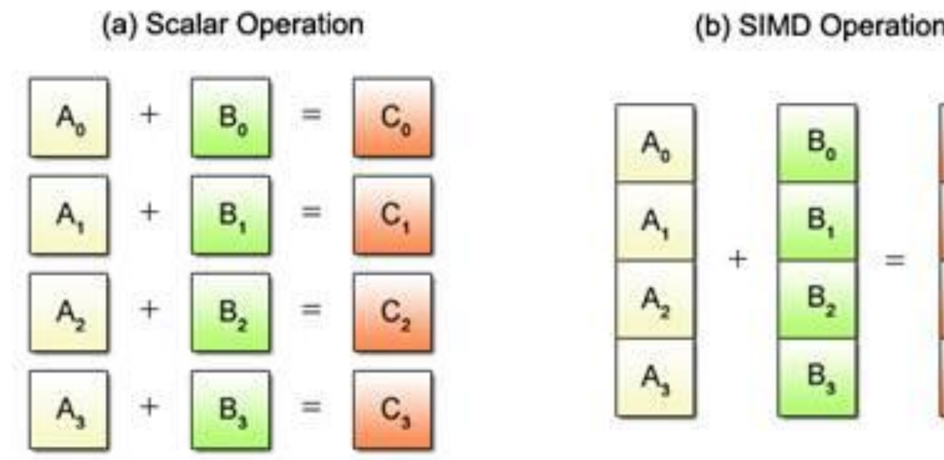


```
int inputNumbers[1000];
int results[4];

int i,j, finalSum;

for (i=0;i<=3;i++) {
  results[i]=0;
  for (j=0;j<=249;j++) {
    results[i] += inputNumbers[i + j*4];
  }
}

finalSum=0;
for (i=0;i<=3;i++) {
  finalSum+=results[i];
}
```

Streaming SIMD Extensions (SSE)

- SIMD instruction set added to Intel CPUs in 1999
  - SSE1 added eight 128 bit registers where data can be packed into and operated on concurrently with associated instructions

```
result.x = v1.x + v2.x;
result.y = v1.y + v2.y;
result.z = v1.z + v2.z;
result.w = v1.w + v2.w;
```



movaps **xmm0**, **[**v1**]**

v1.x	v1.y	v1.z	v1.w

addps **xmm0**, **[**v2**]**

v1.x+v2.x	v1.y+v2.y	v1.z+v2.z	v1.w+v2.w

SIMD technologies

128 bit

SSE

2 x DP

4 x SP

256 bit

AVX

4 x DP

8 x SP

512 bit

MIC
AVX-512

8 x DP

16 x SP

epcc

THE UNIVERSITY OF EDINBURGH

Automatic vectorisation

- Compilers will attempt to automatically vectorise your code when compiled with optimisation enabled (–O3 on GCC)
  - With GCC you can get feedback on this using the -ftree-vectorizer-verbose=n flag, where n is 1 to 6 (the higher = more information)

- For single and double precision floating point can instruct the compiler to do this via SSE
  - With gcc using the flags -msse2, -mfpmath=sse
  - Can involve lots of memory to register movements so work experimenting with this flag to see if it is worth it

Manual vectorisation through GCC

- Compiler intrinsics support SSE

The base type

Vector is 16 bytes wide, which is 4 integers

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si v1, v2, result;

result = v1 + v2;
```

Each of these variables contains 4 integer elements

Each element of v1 added to corresponding element of v2 and the result stored in result

Compiler intrinsics with sum example

```
int inputNumbers[1000];

int i,finalSum;

finalSum=0;
for (i=0;i<=999;i++) {
  finalSum+=inputNumbers[i];
}
```

```
#include <emmintrin.h>
…
int inputNumbers[1000] ;

__m128i s, v =_mm_set_epi32(0,0,0,0);

int j, finalSum=0;

for (j=0;j<=999;j+=4) {
  s=_mm_set_epi32(inputNumbers[j], inputNumbers[j+1],
          inputNumbers[j+2], inputNumbers[j+3]);
  v+=s;
}

for (j=0;j<=3;j++) {
  finalSum+=((int*)&v)[j];
}
```

Compiled with flags -msse, -msse2, -march=native

OpenMP 4.0 SIMD

```
int inputNumbers[1000];
int i,finalSum=0;
#pragma omp simd reduction(+:finalSum)
for (i=0;i<=999;i++) {
  finalSum+=inputNumbers[i];
}
```

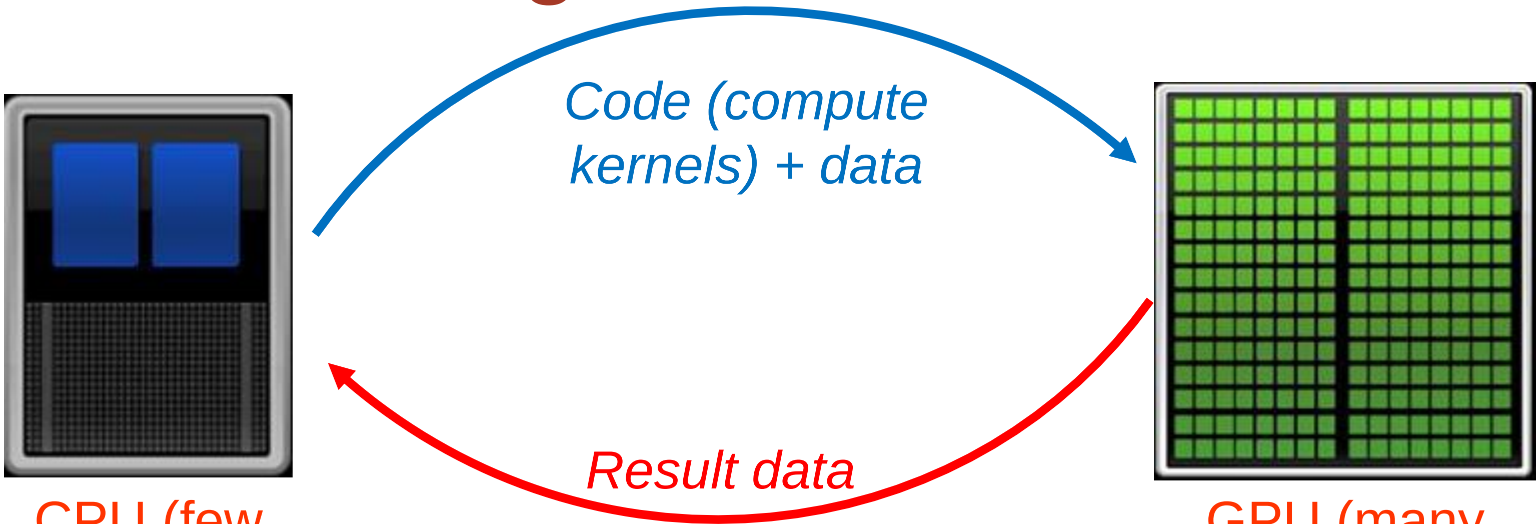
- The SIMD directive means that iterations of the loop can be executed by the SIMD lanes available to the thread.

- Can combine with the *for* directive to split iterations across threads and then across SIMD lanes
  - *The schedule should be a multiple of the SIMD length*

```
int inputNumbers[1000];
int i,finalSum=0;
#pragma omp for simd \
reduction(+:finalSum) schedule (static, 4)
for (i=0;i<=999;i++) {
  finalSum+=inputNumbers[i];
}
```

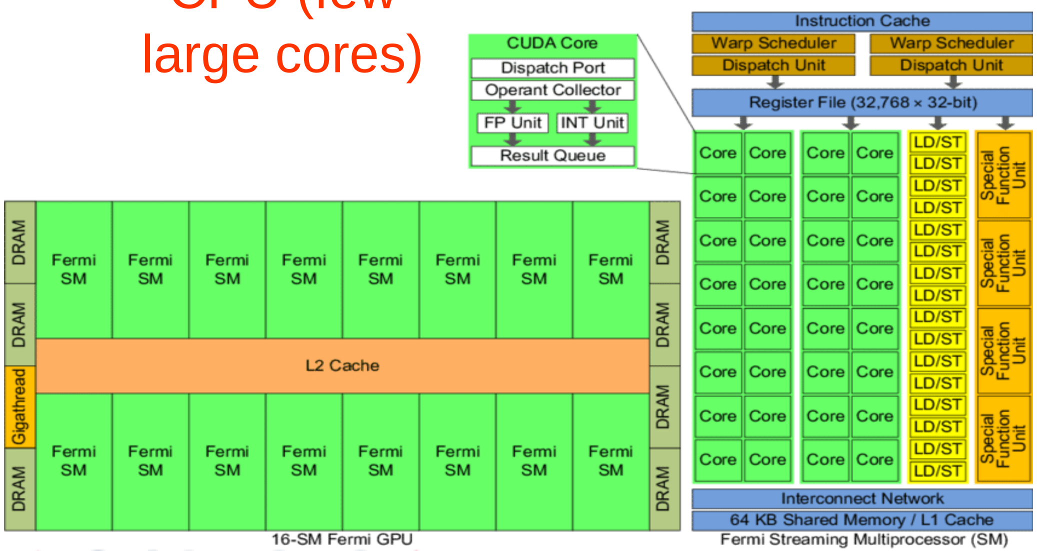
https://doc.itc.rwth-aachen.de/download/attachments/28344675/SIMD+Vectorization+with+OpenMP.PDF

GPUs as a big vector machine



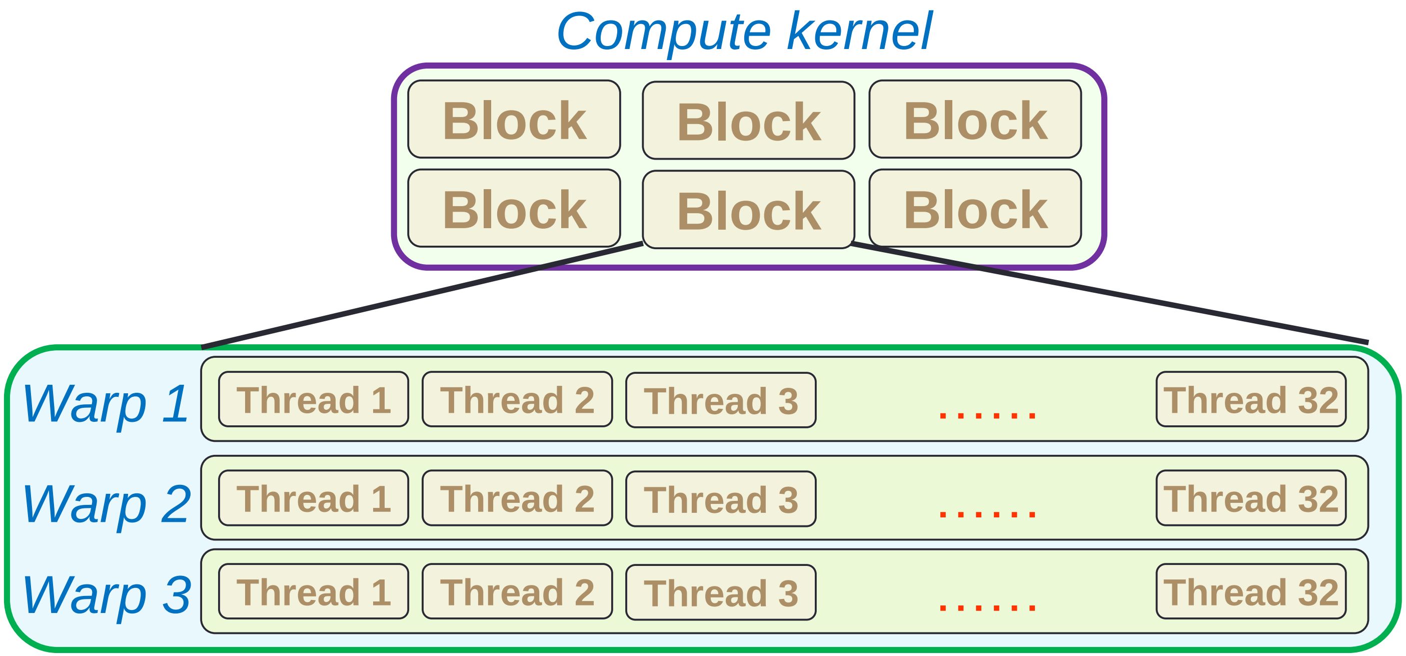
CPU (few large cores)

GPU (many simple cores)



- Use GPU for floating point intensive calculations
- Use CPU for everything else
- Single Instruction Multiple Thread (SIMT)

Kernels, Blocks, Warps and Threads



- 32 threads per warp which are mapped to SMs for execution
  - Each thread executes on a CUDA core which are themselves pipelined
- Each thread of the warp executing on a CUDA core must be doing the same instruction, just on different data
  - Keeps electronics simple, warps can be paused and interleaved

Key performance factors

1. How quickly you can transfer data to & from the GPU
   - Parallel overhead
2. The amount of time the CPU and/or GPU will be idle
   - Wasted resources/load imbalance

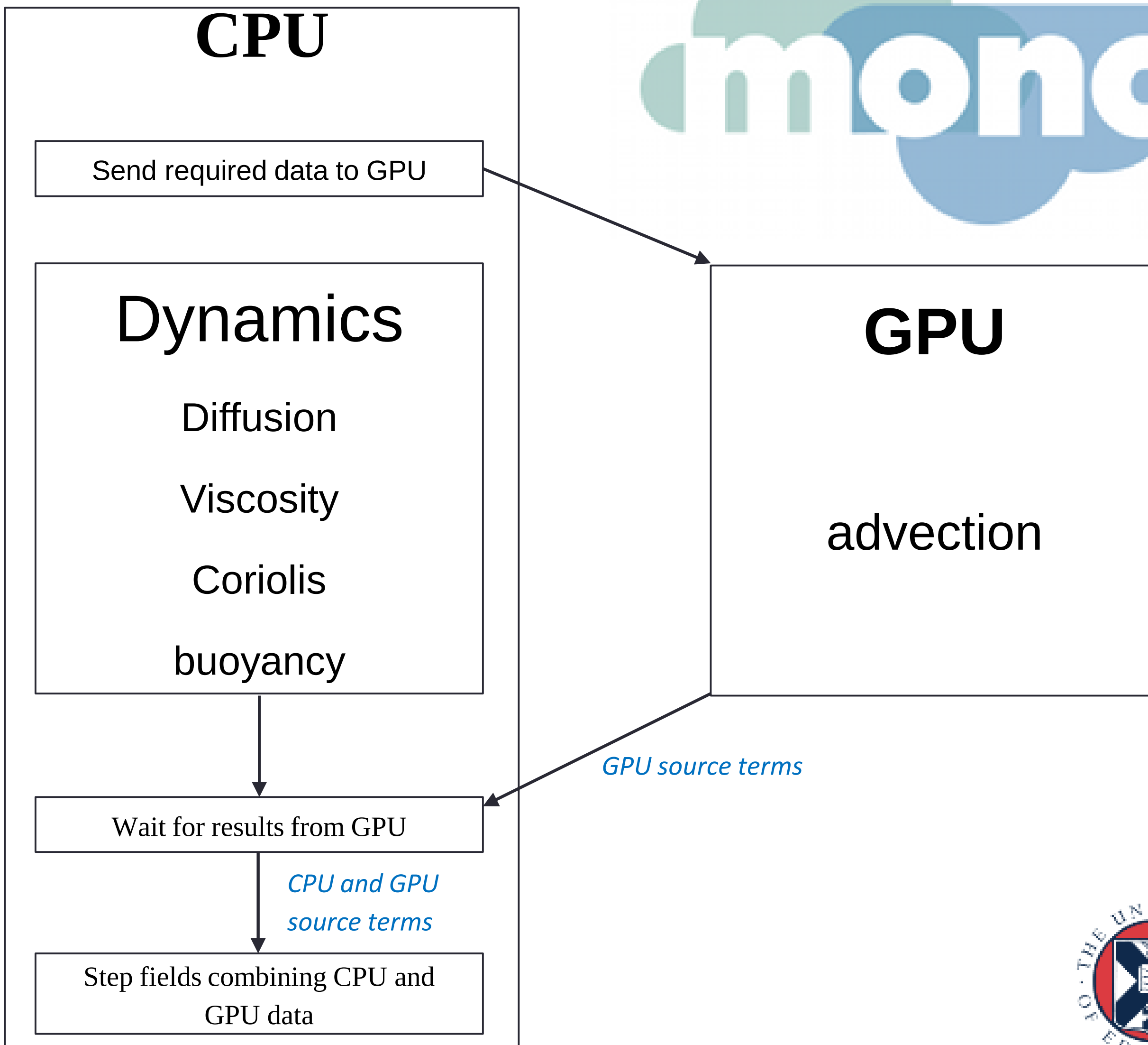
3. How well your code takes advantage of the GPU architecture
   - Keeping the floating point engine busy!

Tesla Products	Tesla P100	Tesla K80	Tesla K40	Tesla M40
GPU	GP100 (Pascal)	2 x GK210 (Kepler)	GK110 (Kepler)	GM200 (Maxwell)
SMs	56	26 (13 per GPU)	15	24
CUDA cores	3840	4992 (2 x 2496)	2880	3072
Base Clock	1328 MHz	560 MHz	745 MHz	948 MHz
GPU Boost Clock	1480 MHz	875 MHz	810/875 MHz	1114 MHz
Peak Double Precision	5.3 TFLOPS	2.91 TFLOPS	1.68 TFLOPS	.2 TFLOPS
Peak Single Precision	10.6 TFLOPS	8.73 TFLOPS	5.04 TFLOPS	7 TFLOPS
Memory Interface	4096-bit HBM2	2 x 384-bit GDDR5	384-bit GDDR5	384-bit GDDR5
Memory Size	16 GB	24GB (12GB per GPU)	12 GB	24 GB
Peak Bandwidth	720 GB/s	480 GB/s (240 GB/s per GPU)	288 GB/s	288 GB/sec
TDP	300 Watts	300 Watts	235 Watts	250 Watts
Transistors	15.3 billion	2 x 7.1 billion	7.1 billion	8 billion
GPU Die Size	610 mm²	2 x 561mm²	551 mm²	601 mm²
Manufacturing Process	16-nm	28-nm	28-nm	28-nm

Porting step	Million pairs/s
Initial MPI+OpenMP	250
Initial OpenACC	37
Optimised data transfer	61
Lattice data kept on GPU	839
Memory access pattern optimised for GPU	1190
Concurrency with streams	1270
Vectorised halo data movement	1812

Example: Modelling the atmosphere

- Data transfer is asynchronous
- Constants copied across only once on model initialisation
- Share data between GPU kernels
  - Wind in x,y,z is common to all

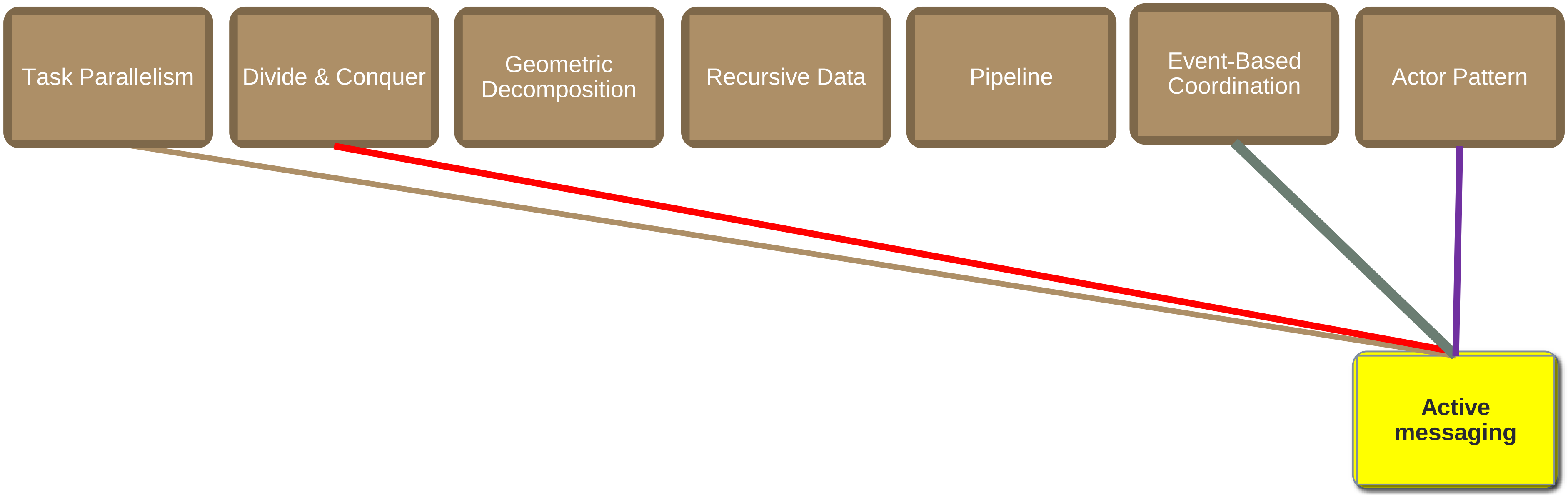


Vectorisation - summary

- Parallelism at multiple levels
  - Instruction level, core level, processor level, node level
  - Significant performance improvements can be obtained by leveraging vectorisation correctly
  - Many compilers will do this automatically for you, but not all compilers are created equally!
  - Technologies such as OpenMP and OpenACC (for GPUs) make this look similar to loop parallelism

- Viewing GPUs as SIMD engines
  - Need to keep them feed with calculations to work on
  - They work best doing floating point arithmetic
  - Need to consider how to keep the CPU and GPU busy at the same time

Active messaging: The Problem



- Active messaging is an *Implementation Strategy*
- The Problem: We want to run multiple tasks, which are driven by irregular interactions, on a UE. How can we best structure our code to support this?

Example problem

- I am running a code with lots of tasks per UE
  - There are lots of tasks (e.g. function calls) that I have available to run on the UE and-so don't want to block for communications. However my communications are irregular and I need to work with values I receive.

```
a=receive(1);
calculate(a);
```

```
handle=nonblocking_receive(1);
while (!test(handle)) {
   Do some other work
}
calculate(a);
```

- This is OK but relies on being able to find some other work to do and carry lots of request handles around
  - Might not be possible, or with irregular & unpredictable communications might be difficult to structure code generally to support this

Active messaging

- The arrival of a message will activate some handling block of code on the target UE (also known as a *callback*)

```
send(data, target rank, unique identifier);
register_recv(callback, source rank, unique identifier);
```

- *The unique identifier* (UUID) is used to match the message with a specific handler
- The callback function will typically receive the data and metadata (such as amount of data, type etc.)
- Sending is either blocking or non-blocking
- The receive call is non-blocking

```
send(data, 1, "hello");
```

UE 0

UE 1

```
register_recv(calculate, 0, "hello");

void calculate(data, metadata) {
   .........
}
```

Active messaging

- Called *active messaging* as messages explicitly activate the block of code which will handle them
  - Some or all of the code will be structured around these handlers
  - Callback handlers might persist (i.e. can be called for many different messages) or transitory (once called they are deregistered.)

- Implementation choice between running handlers concurrently or sequentially
  - When a message arrives do we kick a UE off (i.e. a thread from a pool) which calls the handler
  - Or are messages queued up and processed one at a time?
- If you run handlers concurrently you will need to protect shared data shared between them (*shared data pattern*.)

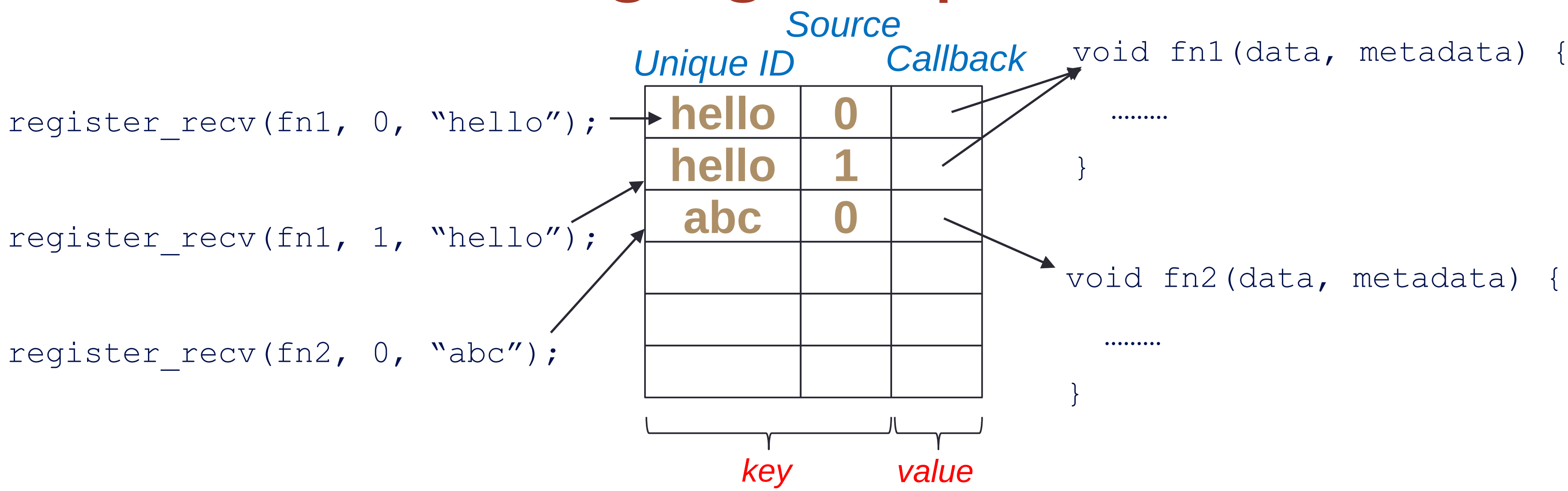
Supports collective messaging too

The callback routine *Value on each process to use* *Operation* *Root* *Unique ID*

```
register_reduce(my_handler, my_value, "sum", 0, "my_reduction1");


void my_handler(data, metadata) {

    .........

}
```

- In this case each process issues a reduction, *my_handler* is then executed on process 0 with the resulting value
  - Callback is only executed on process 0 once every single process has issued this call and the reduction is completed
  - The callback routine could be NULL on other processes
- Crucially the UUIDs determine what collective messages match rather than the issue order
  - This provides greater flexibility for irregular applications where codes might issue collective messages in different orders.

Active messaging - implementation



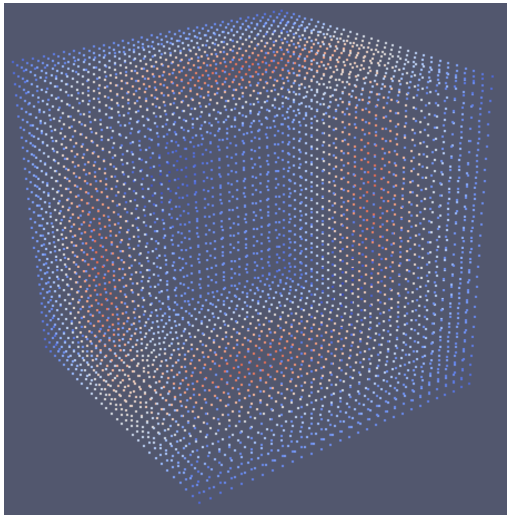
- Have a map style structure where they key is a combination of the unique identifier and the source rank, the value is a pointer to the appropriate callback function
- Behind the scenes you poll for a messages, from this extract the unique ID and use this in combination with the source rank to find the appropriate callback handler function to execute
  - The rest of the message is then split up to extract the data and any other metadata

Active messaging - implementation

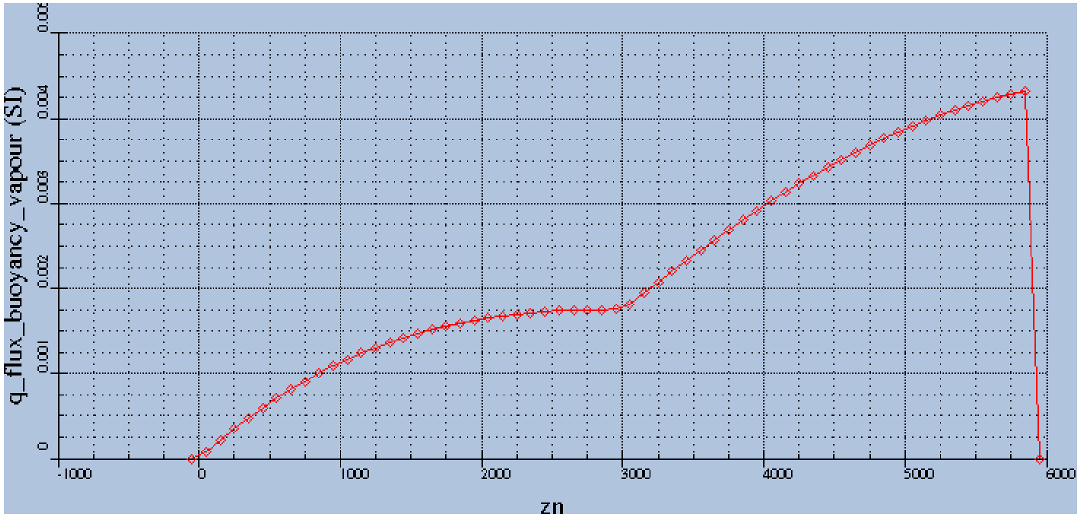
- Can build this on top of communication technologies like MPI
  - When sending package the data and metadata (unique ID etc) up and send as type MPI_BYTE
  - On the receiver side can probe for a messages and extract the message size (and source) from the status, allocate memory and then physical receive data (via MPI_Recv.)
    - Might be driven by a thread continually polling for incoming data

- Some implementation challenges
  - What if we have not yet registered a receive handler for a specific message but this message has arrived? – Need to store unmatched messages
  - When should we terminate? –when all UEs are idle, there is no data in flight and no messages are outstanding

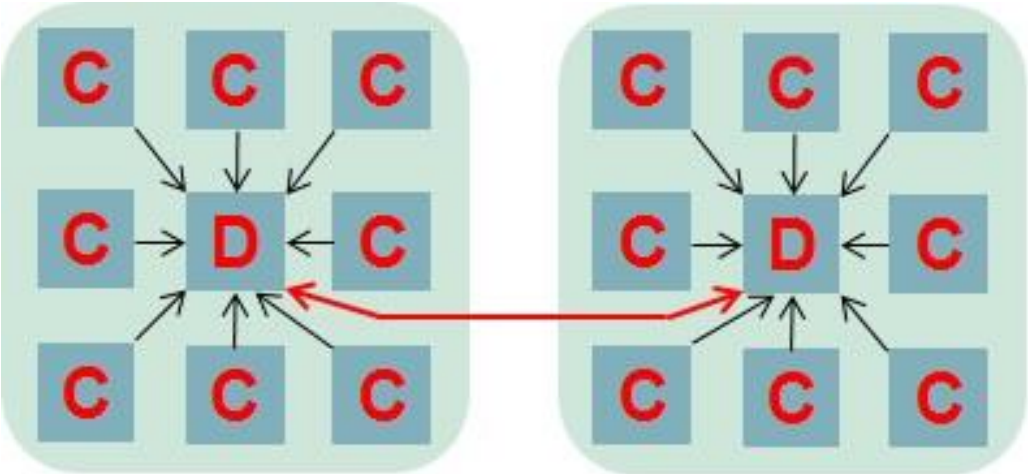
Example: In-situ data analytics

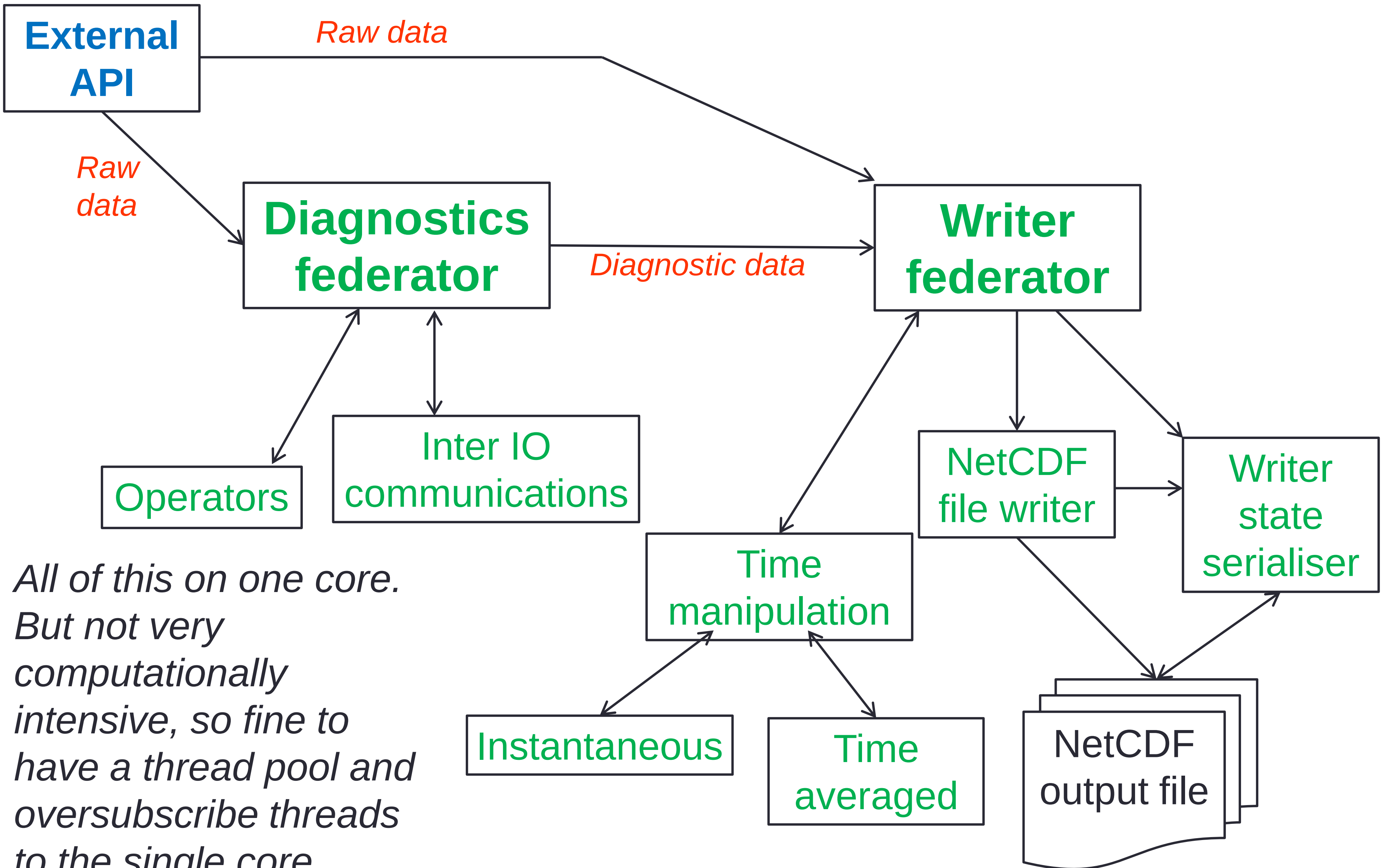


Prognostics



Diagnostics





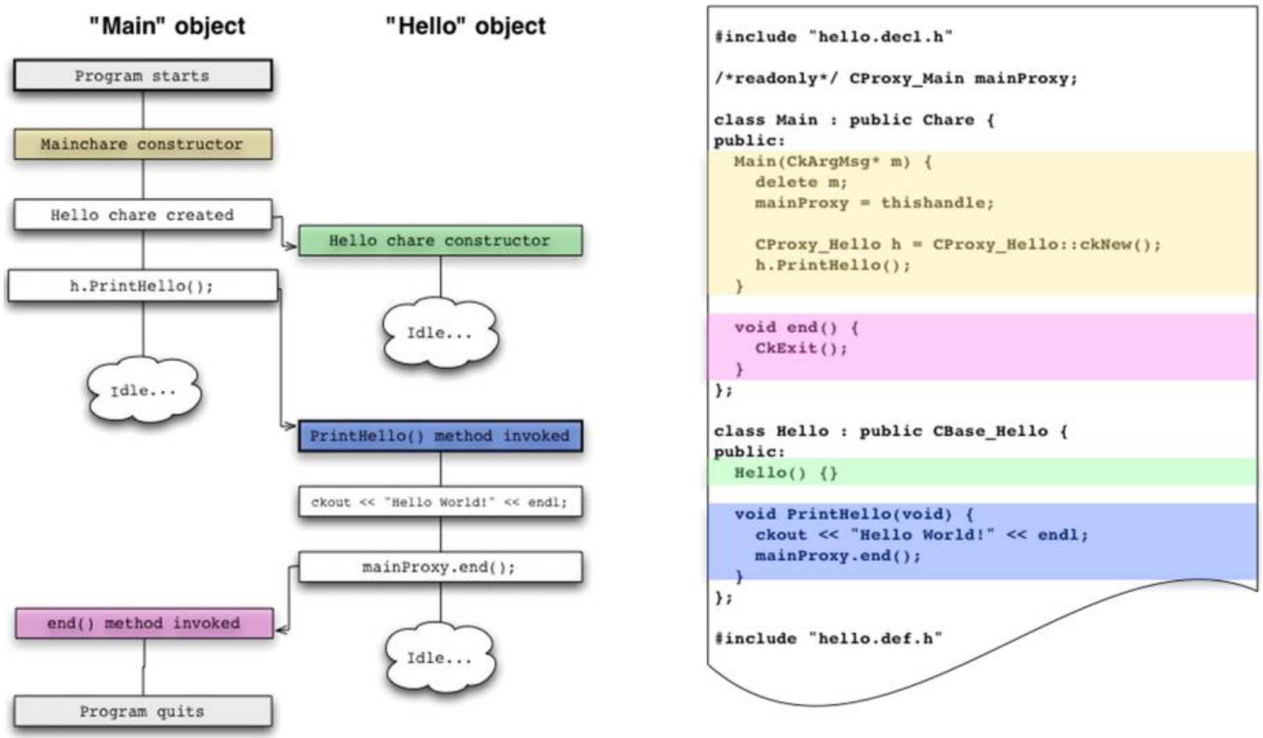
All of this on one core. But not very computationally intensive, so fine to have a thread pool and oversubscribe threads to the single core

Active messaging technologies

- In other fields active messaging is fairly popular
  - Remote Procedure Call (RPC) is a concrete example of this such as Java’s Remote Method Invocation (RMI)
- Not so much in HPC but Charm++ is one example technology
  - Built on C++, the programmer expresses their program components as parallel objects called *chares*
  - The programmer can call methods on these chares held on other processes, which is effectively an active message to execute that method remotely with the provided arguments in a thread
  - As methods in a chare can share object data, by default only one method can be active at any one time (*one at a time concurrency protection – see shared data lecture.)*
  - NAMD, a popular molecular dynamics package is written in Charm++

Charm++ example

Taken from http://charm.cs.illinois.edu/research/charm



- Programmer must rewrite their code in C++ and this chares approach
  - An additional .ci file must be written that defines a proxy for each object and feeds into their compiler
- One at a time concurrently is limiting, can disable this but then is entirely up to the programmer to manage concurrency

Active messaging - Summary

- This way of structuring the communications can provide additional flexibility
  - Can be helpful when you have very many, asynchronous and different messages which you want to process in different ways
  - Using the unique identifier to match against handling logic means you can kick off lots of communications without worrying too much about the ordering in which they will arrive
- Structuring the code in this manner can help organise the concurrency
  - Especially if you allow for multiple handlers to execute concurrently
  - Each handler can be viewed as a task, driven by the arrival of data. But it gets more challenging when these handlers need to interact or work with some shared data
  - There are existing programming technologies, but none are mature