Level 7 Advanced Computational Physics – Lecture 7

Dr. Simon Hanna

October 19, 2021

9 Code Optimisation

9.1 Data Dependencies

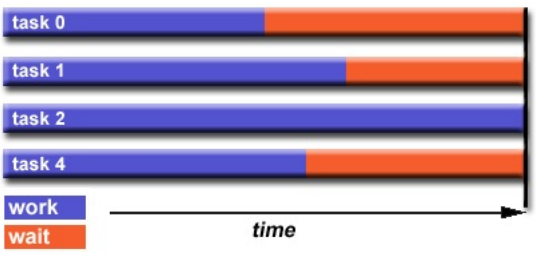
- A dependence exists between program statements when order of execution affects the results of the program.
- A data dependence results from multiple use of the same data element by different tasks.
- Dependencies are one of the main inhibitors of parallelism.
- Loop carried data dependence:

```
for j in prange(N):
    a[j] = a[j-1] * 2.0
```

  `a(j-1)` must be computed before `a(j)`, therefore `a(j)` exhibits data dependency on `a(j-1)`. Parallelism is inhibited.
- Loop carried dependencies are particularly important because loops are most common target of parallelization efforts.

9.2 Load Balancing

- The practice of distributing approximately equal amounts of work among tasks, so all tasks are kept equally busy all of the time.
- A minimization of task idleness.
- Important for optimising performance. E.g., if all tasks subject to a barrier synchronization, slowest task will determine overall performance:

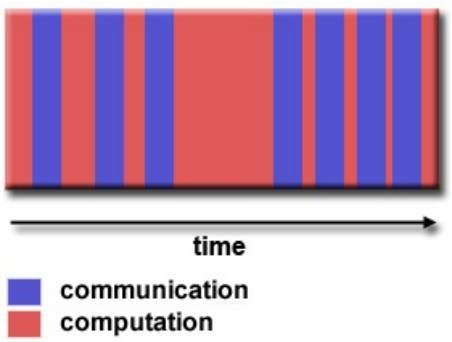


There are two main ways to achieve load balance:

1. Equally partition the work each task receives (easy)
   - Divide arrays/matrices evenly between tasks;
   - Evenly distribute loop iterations across tasks;
2. Use dynamic work assignment (hard)
3. Certain problems result in load imbalance even when data appear evenly distributed:
   - Sparse arrays – some tasks will have data to work on while others have mostly zeros.
   - Adaptive grid methods – some tasks may refine their mesh while others don't.
   - N-body simulation domain decomposition – some particles may migrate between task domains (see below).
   - Use a task scheduler/task pool approach to allocate the work as each task finishes.

9.3 Granularity

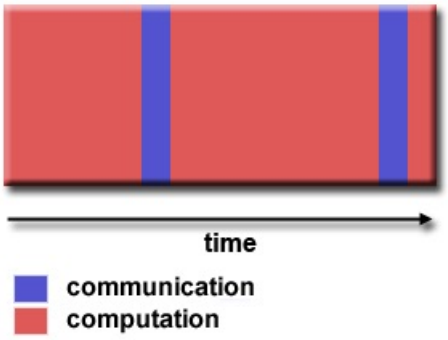
- Effectively the computation / communication ratio.
- Fine-grain Parallelism:



  – Relatively small amounts of computational work between communication events

  - Low computation to communication ratio
  - Facilitates load balancing
  - But, high communication overhead and less opportunity for performance enhancement
  - If granularity is too fine communication and synchronization overhead may dominate: solution may be to make the model larger.

- Coarse-grain Parallelism:



  - Relatively large amounts of computation between communication/synchronization events
  - High computation to communication ratio
  - More opportunity for performance increase
  - Harder to load balance efficiently

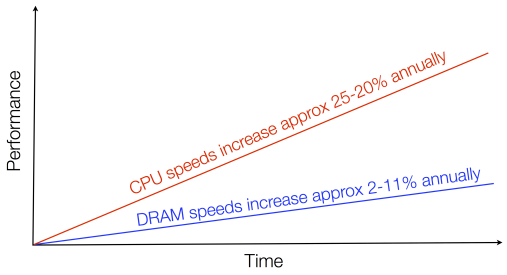
- Choice of granularity depends on algorithm and available hardware

9.4 Input/Output

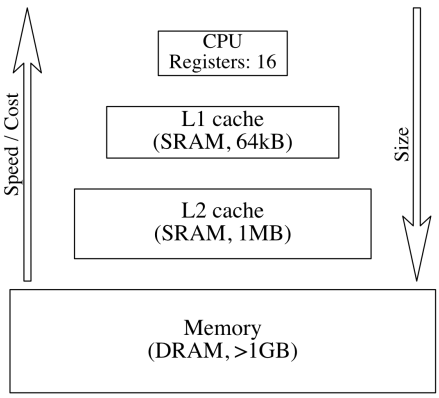
- The Bad News:
  - I/O operations generally inhibit parallelism.
  - I/O operations orders of magnitude slower than memory operations.
  - Parallel I/O systems rarely available.
  - If all tasks see the same file, write operations can cause overwriting.
  - File reading can be affected by ability to handle simultaneous reads.
  - I/O over a network can cause severe bottlenecks.
- A few rules to follow:
  - Reduce I/O as much as possible
  - Write large chunks of data rather than small chunks – this is usually more efficient.
  - Working with fewer, larger files is more efficient than handling many small files.
  - If possible, confine I/O to serial portions of program, under control of a single task e.g. the master.

9.5 Memory Handling

9.5.1 Memory Systems



- Memory is too slow to keep up with the processor, and is not catching up:
  - 100–1000 cycles latency before data arrives
  - Data stream may deliver 1/4 floating point number per clock cycle, whereas processor wants 2 or 3...
- At considerable cost it's possible to build faster memory
- Cache is small amount of fast memory



- Memory is divided into different levels:

  - Registers
  - Caches
  - Main Memory

  May be 3rd level cache as well.

- Memory is accessed through the hierarchy
  - registers where possible
  - ... then the caches
  - ... then main memory

- Transfer of data between memory and caches is under hardware control. But, we can exploit our knowledge of cache sizes to make code run faster.
- Two important concepts:

  Latency: The time it takes to initiate the transfer of a piece of memory

  Bandwith: The data rate that can be sustained once the transfer is started; units are B/sec (MB/sec, GB/sec, etc.)

- Typical relative latencies and bandwidths:

	Latency	Bandwidth
Registers	0	
L1 cache	$\sim 5CP$	$\sim 2W/CP$
L2 cache	$\sim 15CP$	$\sim 1W/CP$
Memory	$\sim 300CP$	$\sim 0.25W/CP$
Dist. mem. (disks etc)	$\sim 10000CP$	$\sim 0.01W/CP$

  (W/CP = words per clock period)

- Hard drives, even SSDs, are very slow – forget about them except for very occasional i/o

9.5.2 Data Caches

- Sit between the CPU Registers and main memory
- L1 Cache: Data cache closest to registers
- L2 Cache: Secondary data cache, stores both data and instructions
  - Data from L2 has to go through L1 to registers
  - L2 is 10 to 100 times larger than L1
  - Some systems have an L3 cache, $\sim 10\times$ larger than L2
- Cache line
  - The smallest unit of data transferred between main memory and the caches (or between levels of cache)
  - N sequentially-stored, multi-byte words (usually N=8 or 16).
  - Every cache has its own line size
  - If you request one word on a cache line, you get the whole line
  - make sure to use the other items, you've paid for them in bandwidth
  - Sequential access good, "strided" access (every n-th) ok, **random access bad**
  - For example, the vector 'dot' product can be very cache-efficient as it requires access of adjacent memory locations.
- Multi-core chips
  - Cores typically have separate L1, shared L2 (or L3) cache i.e. a hybrid shared/distributed model
  - Cache coherency can be a problem: conflicting access to duplicated cache lines i.e. if cores 1 and 2 both have copies of the same block of memory in their L1 caches, which is correct? Hardware will keep track, but there may be performance hit because of it.
  - Can be an issue when using a shared memory model (i.e. not with MPI). Need to use private variables to mitigate some of the issues with cache conflicts and race-conditions.

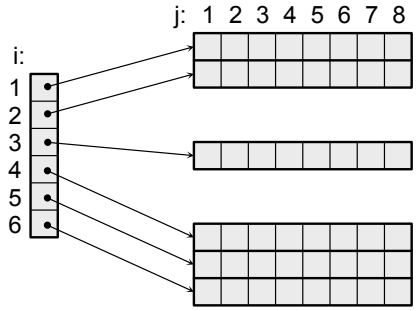
9.5.3 Efficient memory access

- As noted above, sequential memory access is preferable to take best advantage of processor caches.
- This is most easily achieved through use of arrays.
- For 1D arrays, consecutive elements will be stored in consecutive memory locations, as required.
- For 2D or higher dimensional arrays, the situation is more complicated:

  python lists / C arrays declared with malloc(): The first dimension will actually contain a reference (pointer) to the location of the second dimension, which will be contiguous

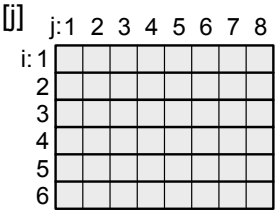
in memory. Thus the elements of each row will be contiguous, but the rows could be stored in different parts of the memory.

A[i][j]



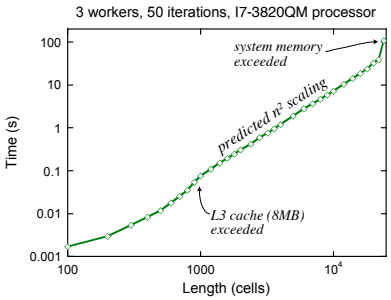
NumPy arrays / C arrays declared without malloc(): The data will be stored in contiguous memory locations in a row (first index) major arrangement.

A[i][j]



Fortran arrays / NumPy arrays declared with Fortran ordering: The data will be stored in contiguous memory locations in a column (second index) major arrangement.

- For efficient memory access (consecutive locations), always iterate through the columns (right most index) in your innermost loops for C or Python, but through the rows (left most index) for Fortran.
- Contiguous memory is essential for data transfer using MPI or mpi4py. If the 2D array is not contiguous, consider transferring individual rows separately.
- To guarantee contiguous memory allocation, consider replacing a 2D array with a 1D array and performing the index algebra yourself.
  - e.g. 1D location = i x (length_of_row) + j
  - in C, this is very efficient as you only need a single memory access rather than separate accesses for each dimension
  - in Python this can be slow due to performing variable typing on i, j and length_of_row for each array access
- NumPy arrays storing simple variables i.e. floats or integers, will always be contiguous.
- Be aware of the amount of memory you require to store your array, compared to the amount available on your system. Exceeding the available RAM on your system can lead to "swapping" where memory is exchanged with data on your hard drive, or data compression. In either case your program will slow down; in extreme cases the program will crash.



- Plot showing effect of data size when running the mpi_heat2d.c program.
- For small models, there is benefit from fitting model within L3 cache, which reduces run times.
- For largest model, data size is many GB and the system is actively compressing the data during the simulation.

10 Vectorization

- Covers a range of techniques that ultimately gain performance by running the same operation repeatedly.

10.1 Vector machines

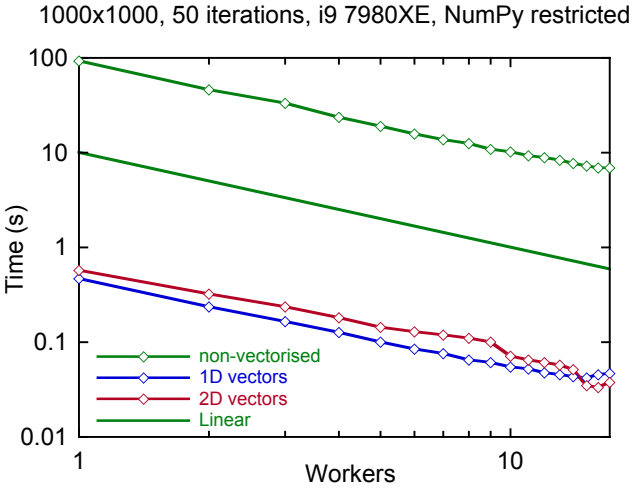
- Old fashioned machines, no longer built, but were very efficient at applying same operation sequentially to block of memory due to pipelining within processor.
- Technique still used in aspects of modern chip design but now referred to as pipelining – just one of many techniques put together to squeeze performance out of silicon.
- Modern compilers may exploit pipelining if the code is written in an obviously vectorizable manner.

10.2 Vectorization in NumPy

- A feature of Python that enables same operations to be applied to all elements of an array, without use of a (slow) `for` loop. e.g.

```
import numpy as np
x = np.linspace(0,1,100)
y = x*x
z = np.sin(x)
```

- Note the need to use the special NumPy math routines e.g. `np.sin(x)`
- NumPy vectorization can be extremely fast e.g. mpi_heat2D.py accelerated by more than two orders of magnitude:



- Work on a subset of an array using slicing syntax:

```
import numpy as np
x = np.linspace(0,1,100)
y = x[0:100:2]*x[100:0:-2]
z = np.sin(x[10:20])
```

  - The y-array will contain 50 elements made up of products of two slices from the x-array.
  - Where multiple arrays are being combined, it is essential that these are the same size and shape.
  - The z-array will contain only 10 elements.

- Example: mpi_heat2D.py without vectorisation:

```
def update(start, end, ny, u1, u2):
    for ix in range(start,end+1):
        for iy in range(1,ny-1):
            u2[ix,iy] =
                (1-2*(Parms_Cx+Parms_Cy)) * u1[ix,iy] +
                Parms_Cx * (u1[ix+1,iy] + u1[ix-1,iy]) +
                Parms_Cy * (u1[ix,iy-1] + u1[ix,iy+1])
```

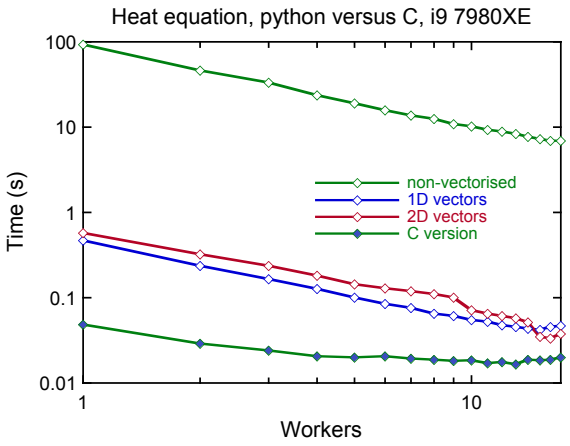
- mpi_heat2D.py with 1D vectorisation:

```
def update(start, end, ny, u1, u2):
    for ix in range(start,end+1):
        u2[ix,1:ny-1] =
            (1-2*(Parms_Cx+Parms_Cy)) * u1[ix,1:ny-1] +
            Parms_Cx * (u1[ix+1,1:ny-1] + u1[ix-1,1:ny-1]) +
            Parms_Cy * (u1[ix,0:ny-2] + u1[ix,2:ny])
```

- mpi_heat2D.py with 2D vectorisation:

```
def update(start, end, ny, u1, u2):
    u2[start:end,1:ny-1] =
        (1-2*(Parms_Cx+Parms_Cy)) * u1[start:end,1:ny-1] +
        Parms_Cx * (u1[start+1:end+1,1:ny-1] +
                    u1[start-1:end-1,1:ny-1]) +
        Parms_Cy * (u1[start:end,0:ny-2] +
                    u1[start:end,2:ny])
```

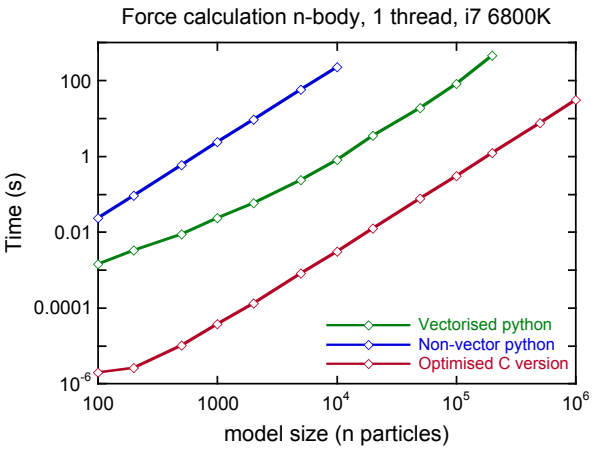
- In some cases NumPy vectorisation can compete with C. For example, in the above heat equation solution, the code is entirely vectorised, and the timings appear like this:



- But, as soon as non-vector code is included, this will enable the C versions of the code to win - e.g. consider an n-body force calculation.

```
/**************************************
/ C Version
/**************************************
for (i=0; i<NPART; i++) {
    double dx, dy, dz, delta, root, factor;
    ax[i] = 0.0;
    ay[i] = 0.0;
    az[i] = 0.0;
    for (j=0; j<NPART; j++) {
        dx = x[j]-x[i];
        dy = y[j]-y[i];
        dz = z[j]-z[i];
        delta = dx*dx + dy*dy + dz*dz + EPS;
        root = sqrt(delta);
        factor = m[j]/(delta*root);
        ax[i] += factor*dx;
        ay[i] += factor*dy;
        az[i] += factor*dz;
    }
    ax[i] *= -BIGG;
    ay[i] *= -BIGG;
    az[i] *= -BIGG;
}
```

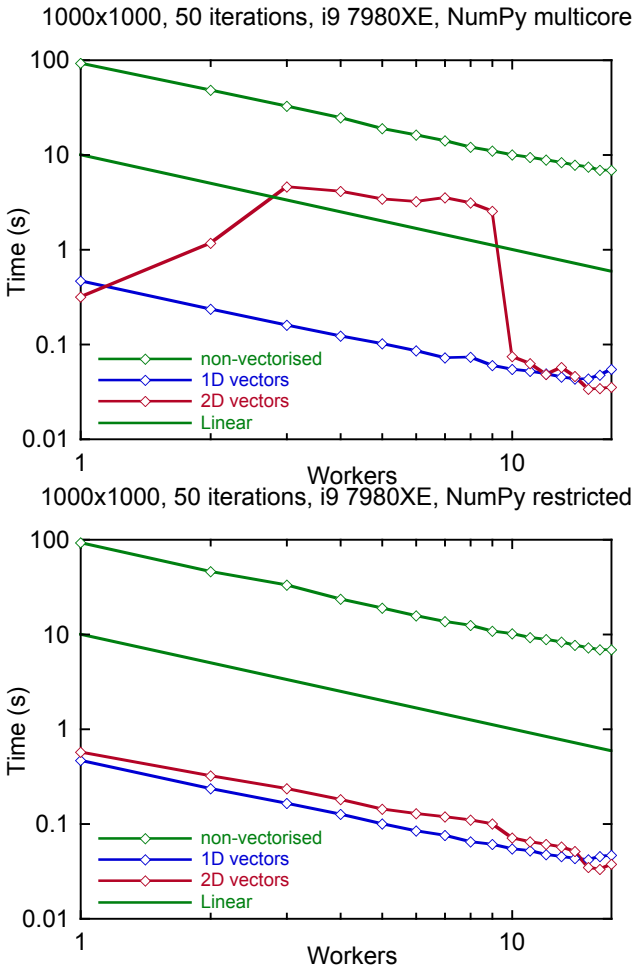
```
/*************************************
/ Python Version
/*************************************
for i in range(NPART):
    ax[i] = 0.0
    ay[i] = 0.0
    az[i] = 0.0
    for j in range(NPART):
        dx = x[j]-x[i]
        dy = y[j]-y[i]
        dz = z[j]-z[i]
        delta = dx*dx + dy*dy + dz*dz + EPS
        root = np.sqrt(delta)
        factor = m[j]/(delta*root)
        ax[i] += factor*dx
        ay[i] += factor*dy
        az[i] += factor*dz
    ax[i] *= -BIGG
    ay[i] *= -BIGG
    az[i] *= -BIGG

/*************************************
/ Python Version Vectorised
/*************************************
def accelerations(x0,y0,z0,m0,x,y,z,m):
    dx = x-x0
    dy = y-y0
    dz = z-z0
    delta = dx*dx + dy*dy + dz*dz + EPS
    root = np.sqrt(delta)
    factor = -BIGG*m/(delta*root)
    ax = np.sum(factor*dx)
    ay = np.sum(factor*dy)
    az = np.sum(factor*dz)
    return ax,ay,az
/*************************************
for i in range(NPART):
    ax[i],ay[i],az[i] =
        accelerations(x[i],y[i],z[i],m[i],x,y,z,m)
/*************************************
```



- In n-body calculation, C version is 2 orders faster than vectorised NumPy code, which is 2 orders faster than raw python code.
- Important note: NumPy will tend to grab all the available cores on your system when running. This is fine on your personal machine but:
  - MPI timings will be seriously compromised;
  - The use of multiple cores appears unpredictable and cannot be relied on as a way of accelerating code;
  - On a supercomputer this can cause serious problems including:
    - causing your own code to run slowly;
    - causing other users code to run slowly;
    - suspension of your account...
  - To force NumPy to keep to a single core or thread, insert the following lines of code at the beginning of your program, **before** you import NumPy:

```
import os
os.environ["MKL_NUM_THREADS"] = "1"
os.environ["NUMEXPR_NUM_THREADS"] = "1"
os.environ["OMP_NUM_THREADS"] = "1"
```

The effect is illustrated here:



- Similar strange effects are observed when using many built-in NumPy functions e.g. for multiplying matrices or other linear algebra applications.

10.3 Instruction level vectorization

- Modern 64 bit processors need registers that are at least 64 bits wide to hold a double precision floating point variable. In practice, registers are now wider than this, so that two or more numbers can be held in each register and manipulated simultaneously with a single set of instructions.
- This is known as instruction level parallelism.
- Various approaches are available on different processors e.g.:
  - Streaming SIMD extensions (SSE) available on Intel chips since late 1990s, pack 4 x 32-bit single precision or 2 x 64 bit double precision numbers into a set of 128-bit registers.
  - Advanced vector extensions (AVX, AVX2, AVX-512 etc) on more modern cpus operate on registers that are 256- or 512-bit, and can handle 4 or 8 double precision numbers simultaneously.
  - These Intel extensions are also available on more recent AMD chips.
  - ARM has its NEON extensions for 128-bit registers and has recently introduced scalable vector extensions (SVE), which can operate with registers up to 2048 bits. However register longer than 512 bits are not found in the wild.
- To simplify use of these extensions, special data types and low level function calls (vector intrinsics) are available in many C/C++ compilers. These are also accessible from Fortran (via wrappers) and from Python (using Cython).
- Many compilers use auto-vectorisation i.e. when the raw C/C++ code suggests it, the special instruction codes will be used. Sometimes, changing the C/C++ code can help the compiler to spot where vectorisation might be beneficial. Fortran is particularly good for this. However, generally explicit calling of vector intrinsics will achieve better speeds.
- Numerical libraries often make extensive use of these features, and tend to be hand-optimised for each specific hardware.

Exercise Week 4 – time to catch up

- If you haven't already applied for an account on BlueCrystal, please do so now. Details are given in the current version of the installation notes. Once you have the account, practice running programs on the queue. Some instructions for this are given in the notes; further information is given in tutorial guides on the BlueCrystal website.
- If you are still having problems installing MPI or mpi4py on your own computer, now is the time to pester Dr Hanna until you get it sorted out. Additionally, come along to the drop-in session on Wednesday and we will try to get you sorted out.
- Have a look at the list of possible mini-projects on Blackboard and start thinking about what you might attempt. If you have thoughts about other topics not listed, feel free to discuss them with Dr Hanna or with our demonstrators, Cale & Mike, at the drop-in session.