OpenMP for Computational Scientists

3: Vectorisation and NUMA

Dr Tom Deakin
University of Bristol

Tuesday 1 December, 2020

Previous exercise

Take your parallel 5-point stencil, and implement a reduction:

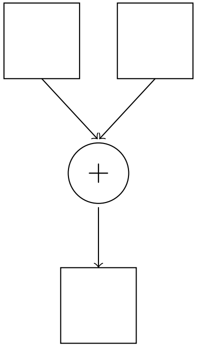
```
total = 0.0
!$omp parallel do collapse(2) reduction(+:total)
do i = 1, nx
    do j = 1, ny
    Atmp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j) + A(i,j-1) + A(i,j+1)) / 5.0
    total = total + Atmp(i,j)
    end do
end do
!$omp end parallel do
```

- Well done if you managed this!
- 5-point stencil is simple, but captures the *essence* of more complicated codes.
- Extension: did anyone try the parallelising the Jacobi solver?

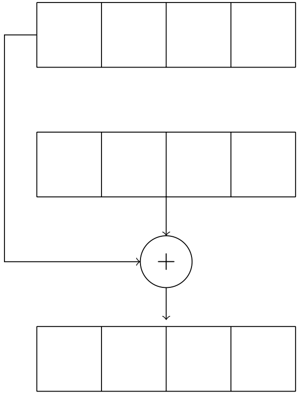
Vectorisation

$$C = A + B$$

Scalar operations



Vector operations



Why vectorise?

- Vectorisation gives you more compute per cycle.
- Hence may increase the FLOP/s rate of the processor.
- Also results in fewer instructions to process (less pressure on instruction decode units).
- Vectors help make good use of the memory hierarchy (often the main benefit).
- Vectorisation helps you write code which has good access patterns to maximise bandwidth.

Auto-vectorisation

- Modern compilers are very good at automatically vectorising your loops.
- Fortran helps as arrays can not alias (overlap), unlike C.
- But compiler needs to be sure it's safe to vectorise.
- Read compiler reports to see if it's already vectorising.
  - Intel: `-qopt-report=5`
  - Cray: `-hlist=a`
  - GNU (old): `-ftree-vectorizer-verbose=2`
  - GNU (new): `-fopt-info-vec`
  - Clang: `-Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize`
- Often the memory access pattern prevents (efficient) auto-vectorisation.

OpenMP SIMD

- Sometimes the compiler needs help in confirming loops are vectorisable.
- OpenMP `simd` constructs give this information.
- Can combine with `parallel do` construct to ensure a parallel vector loop: `omp parallel do simd`
- Generally want to vectorise inner loops and parallelise outer loops.

```
!$omp simd
do i = 1, N
  C(i) = A(i)+B(i)
end do
!$omp end simd
```

SIMD functions

Say you've written an update function to update values in the loop:

```
do i = 1, N
  A(i) = magic_maths(A(i))
end do
```

- The situation gets complicated.
- If the function is small, then likely inlined and loop will auto-vectorise.
- Otherwise need to use the `simd` construct, but need compiler to create a vector version of the function.

```
function magic_maths(value) result(r)
!$omp declare simd(magic_maths)
  implicit none
  real(kind=8) :: value, r
  r = value * value
end function
```

SIMD clauses

- All the usual data-sharing and reduction clauses can be applied.
- `safelen(4)`: distance between iterations where its safe to vectorise.

```
!$omp simd safelen(4)
do i = 1, N-4
  A(i) = A(i) + A(i+4)
end do
!$omp end simd
```

- `simdlen(4)`: preferred iterations to be performed concurrently as a vector. Specifying explicit vector lengths builds in obsolescence to the code as hardware vector lenghts continually change — don't recommend using this clause.

SIMD clauses

- `linear(var)`: variable is private and linear to the loop iterator.

```
!$omp simd linear(j)
do i = 1, N
  j = j + 1
  A(j) = B(i)
end do
!$omp end simd
```

- `aligned(var)`: says the array is aligned (more on this shortly).
- `uniform(var)`: for `declare simd` construct, the variable is the same in all vector lanes.

SIMD summary

- Sometimes need to force the compiler to auto-vectorise (the correct) loop with the `simd` construct.
- As with `parallel`, you are telling the compiler it is safe to vectorise and to ignore its data dependancy analysis.
- Check the compiler report before and after the check it did the right thing!
- Use `declare simd` and appropriate clauses if you need to create vectorised versions of functions.
  - The clauses can give more information to the compiler so it does a better job.

Derived types

2D grid of cells, each cell containing 4 different values.

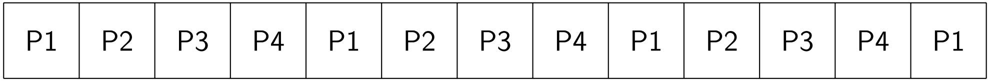
```fortran
type cell
  real(kind=8) :: property1
  real(kind=8) :: property2
  real(kind=8) :: property3
  real(kind=8) :: property4
end type

type(cell), allocatable :: grid(:,:)

do j = 1, ny
  do i = 1, nx
    grid(i,j)%property1 = update_1()
    grid(i,j)%property2 = update_2()
    grid(i,j)%property3 = update_3()
    grid(i,j)%property4 = update_4()
  end do
end do
```

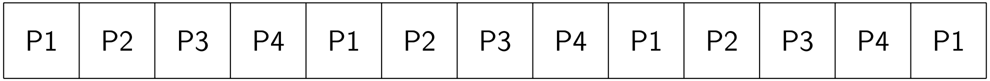
Derived types

- What do Fortran derived types look like in memory?
- Organised as an array of structures.



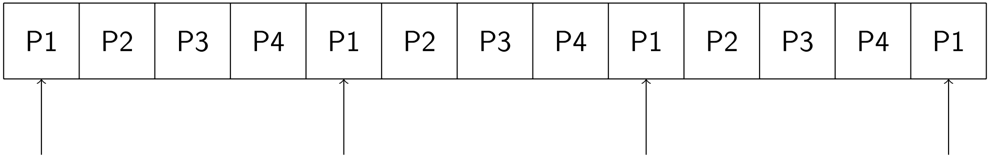
Derived types

- What do Fortran derived types look like in memory?
- Organised as an array of structures.
- What happens when we vectorise our loop over cells?



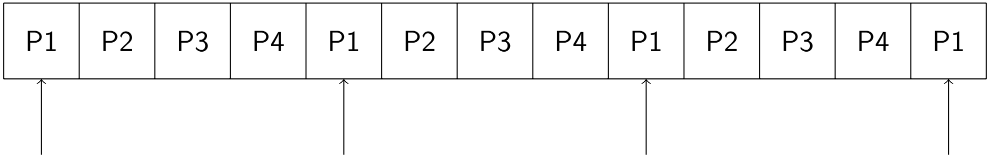
Derived types

- What do Fortran derived types look like in memory?
- Organised as an array of structures.
- What happens when we vectorise our loop over cells?



Derived types

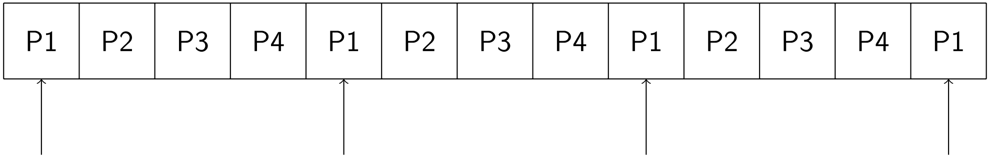
- What do Fortran derived types look like in memory?
- Organised as an array of structures.
- What happens when we vectorise our loop over cells?



- The `property1` values are gathered into a vector register.

Derived types

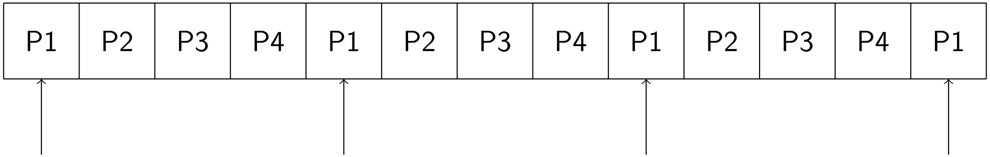
- What do Fortran derived types look like in memory?
- Organised as an array of structures.
- What happens when we vectorise our loop over cells?



- The `property1` values are gathered into a vector register.
- After the computation, the results are scattered back into memory.

Derived types

- What do Fortran derived types look like in memory?
- Organised as an array of structures.
- What happens when we vectorise our loop over cells?



- The `property1` values are gathered into a vector register.
- After the computation, the results are scattered back into memory.
- A cache line is 64 bytes, so only the first two values are on the first cache line.
- Must read two cache lines to fill the vector up.

Structure of arrays

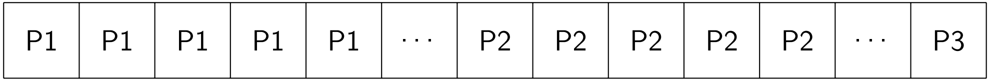
Switch type around to have an array per property.

```fortran
type grid
  real(kind=8), allocatable :: property1(:,:)
  real(kind=8), allocatable :: property2(:,:)
  real(kind=8), allocatable :: property3(:,:)
  real(kind=8), allocatable :: property4(:,:)
end type

do j = 1, ny
  do i = 1, nx
    grid%property1(i,j) = update_1()
    grid%property2(i,j) = update_2()
    grid%property3(i,j) = update_3()
    grid%property4(i,j) = update_4()
  end do
end do
```

Structure of arrays

- Order of data in memory has changed.



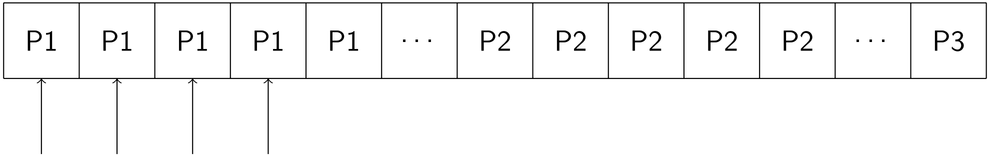
Structure of arrays

- Order of data in memory has changed.
- What happens when we vectorise?

P1	P1	P1	P1	P1	$\cdots$	P2	P2	P2	P2	P2	$\cdots$	P3

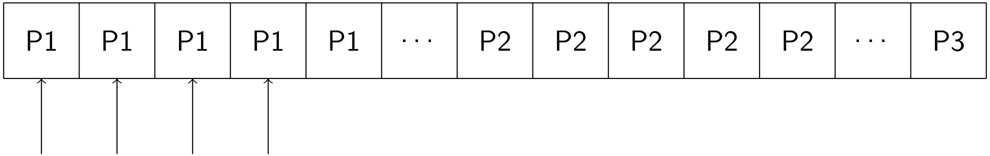
Structure of arrays

- Order of data in memory has changed.
- What happens when we vectorise?



Structure of arrays

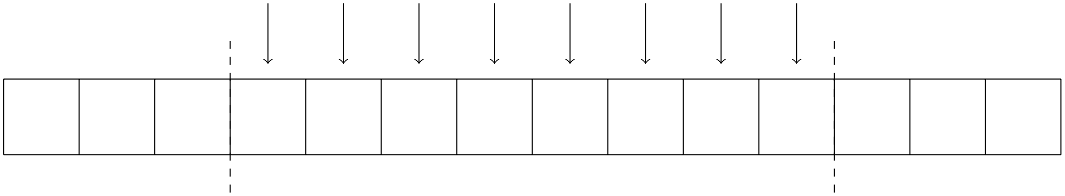
- Order of data in memory has changed.
- What happens when we vectorise?



- Coalesced memory accesses are key for high performance code.
- Adjacent vector lanes read adjacent memory locations.
- A cache line is 64 bytes, so can fill the vector from a single cache line.
- More efficient vectorisation.

Memory access patterns

```
do i = 1, N
  val = A(i)
end do
```

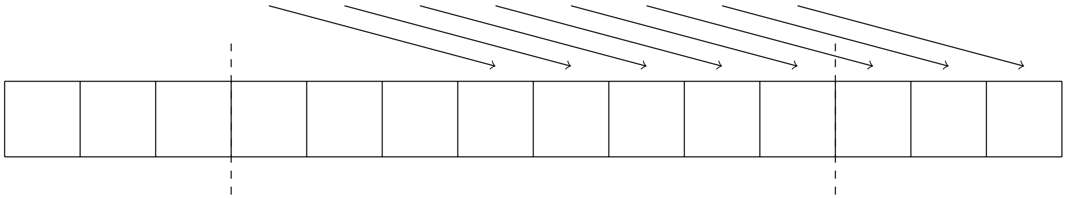


64 byte boundary

- Ideal memory access pattern.
- All access is coalesced.
- Vectors are aligned to cache line boundary.

Memory access patterns

```
do i = 1, N
  val = A(i+3)
end do
```

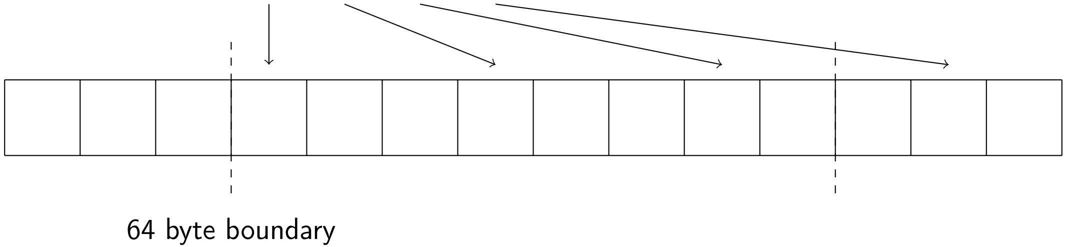


64 byte boundary

- OK memory access pattern.
- All access is coalesced, but split across cache lines.
- Still get good use of cache lines, but not as efficient as aligned version.

Memory access patterns

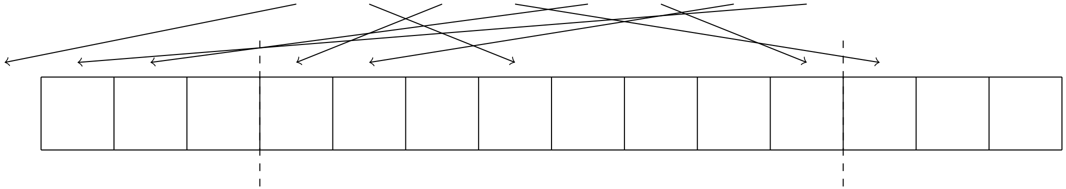
```fortran
do i = 1, N
  val = A(j,i) ! equiv. A(j+3*i)
end do
```



- Strided access results in multiple memory transactions.
- Kills throughput due to poor reuse of cached data.
- Very easy to fall into this trap with multi-dimensional arrays.
- Check your strides!

Memory access patterns

```
do i = 1, N
  val = A(B(i))
end do
```

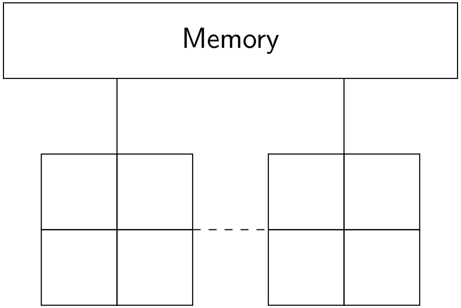


64 byte boundary

- Essentially random access to memory.
- Little reuse of cache lines.
- Unpredictable pattern, so hardware prefetchers won't work efficiently.
- Very challenging!

NUMA Architecture

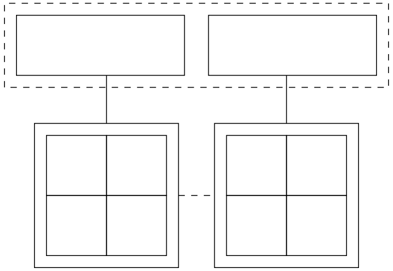
Recall this cartoon of a dual-socket, shared memory system:



All threads (each running on a core) can access the same memory.

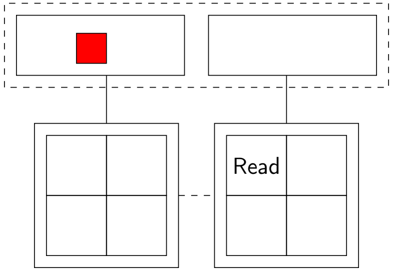
NUMA Architecture

- In reality on a dual-socket system each *socket* is physically connected to half of the memory.
- Still shared memory: all cores can access all the memory.
- A core in the first socket wanting memory attached to the other socket must:
  - Go via the socket-to-socket interconnect.
  - Access memory via the other socket's memory controllers.
- Accessing memory from other socket is slower than access from own socket.



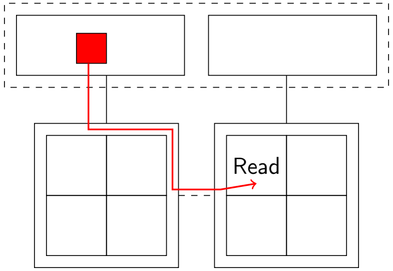
NUMA Architecture

- In reality on a dual-socket system each *socket* is physically connected to half of the memory.
- Still shared memory: all cores can access all the memory.
- A core in the first socket wanting memory attached to the other socket must:
    - Go via the socket-to-socket interconnect.
    - Access memory via the other socket's memory controllers.
- Accessing memory from other socket is slower than access from own socket.



NUMA Architecture

- In reality on a dual-socket system each *socket* is physically connected to half of the memory.
- Still shared memory: all cores can access all the memory.
- A core in the first socket wanting memory attached to the other socket must:
  - Go via the socket-to-socket interconnect.
  - Access memory via the other socket's memory controllers.
- Accessing memory from other socket is slower than access from own socket.



Memory allocation

- What happens when you run `allocate(A(1:N))`?

Memory allocation

- What happens when you run `allocate(A(1:N))`?
- Allocating memory does not necessarily allocate memory!
- Memory is allocated when it's first used (i.e. `A(i) = 1.0`), one *page* at a time.
- OS tends to use a *first touch policy*.
- Memory is allocated in the closest NUMA region to the thread that first touches the data.
- Ideally want threads to use data in local NUMA region to reduce socket-to-socket interconnect transfers.

Taking advantage of first touch

Parallelising your data initialisation routine might mean your main loops go faster!

```fortran
! Allocate and initialise vectors
allocate(A(N), B(N), C(N))
!$omp parallel do
do i = 1, N
  A(i) = 1.0
  B(i) = 2.0
  C(i) = 0.0
end do
!$omp end parallel do

! Vector add
!$omp parallel do
do i = 1, N
  C(i) = A(i) + B(i)
end do
!$omp end parallel do
```

NUMA-aware

- Parallelise your initialisation routines the same way you parallelise the main loops.
- This means each thread touches the same data in initialisation and compute.
- Should reduce the number of remote memory accesses needed and improve run times.
- But, OS is allowed to move threads around cores, and between sockets.
- This will mess up your NUMA aware code!

Pinning threads

- OpenMP gives you the controls to pin threads to specific cores.
- Exposed as *places* and *thread pinning policy* to those places.
- By default there is one place consisting of all the cores.
- Use the `OMP_PROC_BIND` environment variable to set pinning for all `parallel` regions.
- Can use the `proc_bind` clause for control of specific regions, but advise against this.

OMP_PROC_BIND

- `OMP_PROC_BIND=false`: Often the default; threads may move! `proc_bind` clauses ignored.
- `OMP_PROC_BIND=true`: Threads won't move, and follow `proc_bind` clauses or else the implementation default pinning.
- `OMP_PROC_BIND=master`: Threads pinned to same place as master thread.
- `OMP_PROC_BIND=close`: Threads are assigned to places close to the master thread. If `OMP_NUM_THREADS.eq.ncores`: thread 0 will pin to core 0; thread 1 will pin to core 1; etc
- `OMP_PROC_BIND=spread`: Threads are assigned to places "sparsely". If `OMP_NUM_THREADS.eq.ncores`: thread 0 will pin to socket 0 core 0; thread 1 will pin to socket 1 core 0; thread 2 will pin to socket 0 core 1; etc.

Places

- The affinity (policy) defines how threads are assigned to places.
- Places allow you to divide up the hardware resource, so that threads can be assigned to them.
- Default: one place with all cores.
- Use `OMP_PLACES` environment variable to control.
- `OMP_PLACES=thread`: each place is a single hardware thread.
- `OMP_PLACES=cores`: each place is a single core (containing one or more hardware threads).
- `OMP_PLACES=sockets`: each place contains the cores of a single socket.
- Can also use list notation: `OMP_PLACES="{0:4},{4:4},{8:4},{12:4}"`

Thread pinning summary

- In general, going to want to just use `OMP_PROC_BIND=true`.
- Sometimes `spread` or `close` gets better performance.
- Pinning rules can get complicated when there are multiple places, so prefer to use the predefined values.
- Most effective with a NUMA-aware implementation.
- Also helps reduce run-to-run timing variability.
- But must be careful with MPI+OpenMP pinning.

Why combine MPI+OpenMP

- Supercomputers are often constructed with a hierarchical structure:
  - Shared memory nodes connected with a network.
- Need MPI (or similar) to communicate between distributed nodes.
- With multi-core, could just run MPI everywhere (flat MPI).
- But there are advantages to running *hybrid* MPI and OpenMP:
  - Larger fewer messages to take advantage of network bandwidth.
  - Fewer MPI ranks to manage (fewer to synchronise and for collectives).
  - Can avoid memory copies for intra-node communication.
  - Reduced memory footprint.
  - Parallelise other problem dimensions not decomposed with MPI.

Thread support levels

- `MPI_THREAD_SINGLE`
  Only one thread will execute (no threads allowed).
- `MPI_THREAD_FUNNELED`
  May spawn threads, but only the original process may call MPI routines: the one that called `MPI_Init`.
- `MPI_THREAD_SERIALIZED`
  May spawn threads and any thread can make MPI calls, but only one at a time. *Your* responsibility to synchronise.
- `MPI_THREAD_MULTIPLE`
  May spawn threads and any thread can make MPI calls. The MPI library has to deal with being called in parallel.

Remember to make sure ranks still match the MPI communications to avoid deadlock.

Exercise

- Take your parallel 5-point stencil code and optimise it.
- Think about:
  - Memory access patterns
  - Vectorisation
  - NUMA
- Note down the performance differences your optimisations make.
- Calculate the achieved memory bandwidth of your stencil code.
- Extension: consider these optimisaions for the Jacobi solver.