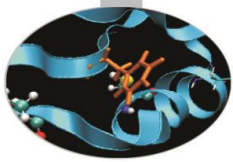
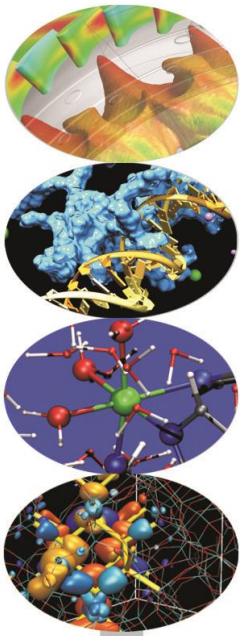
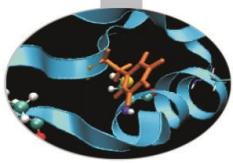


Compilers and Optimisation



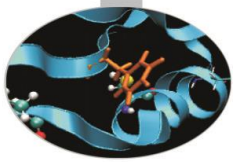
Andrew Emerson, Fabio Affinito
SCAI, Cineca

Contents



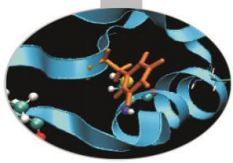
- Introduction
- Compilers and compiling
- Optimisations
  - Loops
  - Vectorisation
  - Other optimisations
- Parallel programs
  - Task and thread affinities
- I/O
- Summary

Introduction



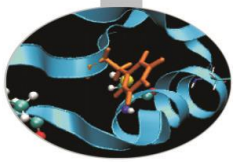
- The hardware components of modern supercomputers are capable providing substantial computing power
- To obtain high performing applications we require:
  - Efficient programming
  - A good understanding of the compilers and how that optimize code for the underlying hardware
  - Tools such as profilers, debuggers, etc, in order to obtain the best performance

The compiler



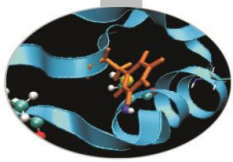
- There are many compilers available and for all computer operating systems (e.g. Linux, Windows or Macintosh).
- As well as free compilers from the GNU project there are also various commercial compilers (e.g. Portland or Intel)
- Some compilers are provided with the hardware (IBM XL for Fermi)

Compilers and interpreters



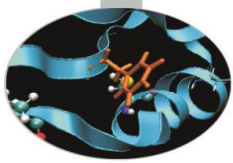
- Interpreted languages
  - The code is “translated” statement-by-statement during the execution
  - Easier on the programmer, modifications can be made quickly but optimisations between different statements (almost) impossible
  - Used for scripting languages (bash, Perl, PHP, ..)
- Compiled languages
  - Entire program is translated before execution
  - Optimisations between different parts of the program possible.
  - HPC languages such as FORTRAN, C and C++

What does the compiler do?



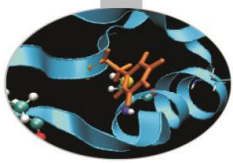
- Translates source code into machine code, if no syntax errors found. Warnings for potential semantic problems.
- Can attempt to optimise the code. Optimisations can be:
  - Language dependent or independent
  - Hardware dependent (e.g. CPU, memory, cache)
- Compilers are very sophisticated software tools but cannot replace human understanding of what the code should do.

Pre-processing, compiling and linking



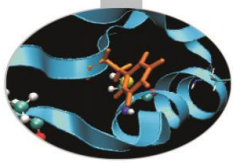
- "Compiling" a program is actually a three stage process:
  1. Pre-processing to replace MACROs (`#define`), code insertions (`#include`), code selections (`#ifdef, #if`). Originally C/C++ but also used in FORTRAN.
  2. Compilation of the source code into object files – organised collections of symbols referring to variables and functions.
  3. Linking of the object files, together with any external libraries to create the executable (if all referred objects are resolved).
- For large projects usual to separate the compiling and linking phases.
- Code optimisations are mainly done during compilation, but how a program is linked may also affect performance (e.g. BG/Q).

Which compiler ?



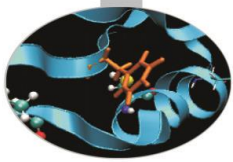
- Common compiler suites include:
  - GNU (gcc, gfortran,...)
  - Intel (icc, icpc, icc)
  - IBM (xlf, xlc, xlC)
  - Portland (pgf90, pgcc, pgCC)
  - LLVM (Clang)
- If I have a choice, which one ?
  - Various things to consider. For performance vendor-specific (e.g xlf on BG/Q, Intel on Intel CPUs) but many tools have been developed with GNU.

What does the compiler do?



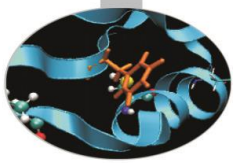
- The compiler can perform many optimisations including:
  - Register allocation
  - Dead and redundant code removal
  - Common subexpression elimination (CSE)
  - Strength reduction (e.g. replacing an exponentiation within a loop with a multiplication)
  - Inlining
  - Loop optimisations such as index reordering, loop pipelining, unrolling, merging
  - Cache blocking

What the compiler does



- What the compiler cannot do:
  - Understand dependencies between data with indirect addressing
  - Non-integer or complex strength reduction
  - Unrolling/Merging/Blocking with
    - Calls to functions or subroutines
    - I/O statements or calls within the code
  - Function in-lining if not explicitly indicated by the programmer
  - Optimize variables with values known only at run-time

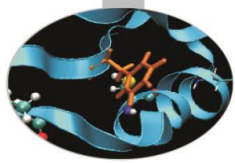
Optimisation options - Intel



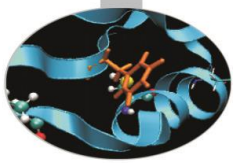
```
icc (or ifort) -O3
```

- Automatic vectorization (use of packed SIMD instructions)
- Loop interchange (for more efficient memory access)
- Loop unrolling (more instruction level parallelism)
- Prefetching (for patterns not recognized by h/w prefetcher)
- Cache blocking (for more reuse of data in cache)
- Loop peeling (allow for misalignment)
- Loop versioning (for loop count; data alignment; runtime dependency tests)
- Memcpy recognition (call Intel's fast memcpy, memset)
- Loop splitting (facilitate vectorization)
- Loop fusion (more efficient vectorization)
- Scalar replacement (reduce array accesses by scalar temps)
- Loop rerolling (enable vectorization)
- Loop reversal (handle dependencies)

Optimisation options



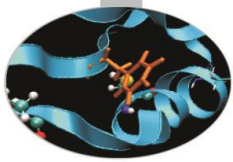
- Compilers give the possibility of specifying optimisation options at compile time, together with the other options.
- These are either general optimisation levels or specific flags related to the underlying hardware.
- Some options can greatly increase the compilation time so one reason for starting with a low optimisation level during code development.



Optimisation levels –common to all HPC compilers

- -O0 : no optimisation, the code is translated literally
- -O1, -O2: local optimisations, compromise between compilation speed, optimisation, code accuracy and executable size (usually default)
- -O3: high optimisation, can alter the semantics of the program (hence not used for debugging)
- -O4 or higher: Aggressive optimisations, depending on hardware.

Can I just leave it to the compiler to optimise my code ?



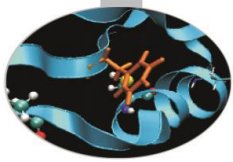
- Example: matrix-matrix multiplication (1024x1024), double precision, FORTRAN.
- Two systems:
  - FERMI: (IBM BG/Q Power A2, 1.6Ghz)
  - PLX: (Xeon Westmere CPUs, 2.4 Ghz)

FERMI xlf

Option	Seconds	MFlops
-O0	65.78	32.6
-O2	7.13	301
-O3	0.78	2735
-O4	55.52	38.7
-O5	0.65	3311

PLX -ifort

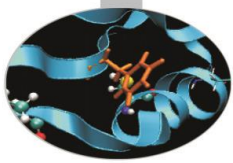
Option	Seconds	MFlops
-O0	8.94	240
-O2	1.41	1514
-O3	0.72	2955
-O4	0.33	6392
-O5	0.32	6623



Can I just leave it to the compiler to optimise my code ?

- To find out what is going on can invoke the `-qreport` option of xlf. It tells us what the compiler is actually doing.
- On Fermi, for –O4 the option tells us that the optimiser follows a different strategy:
  - The compiler recognises the matrix-matrix product and substitutes the code with a call to a library routine __xl_dgemm
  - This is quite slow, particularly compared to the IBM optimised library (ESSL).
  - Intel uses a similar strategy, but uses instead the efficient MKL library
- Moral? Increasing the optimisation level doesn't always increase performance. Must check each time.

Optimisation Reports

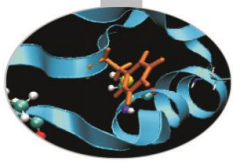


- Compiler dependent. Intel provides various useful options:

```
-opt-report[n] n=0(none),1(min),2(med),3(max)
-opt-report-file<file>
-vec-report[n] n=0(none),1(min),2,3,4,5,6,7(max)
....
```

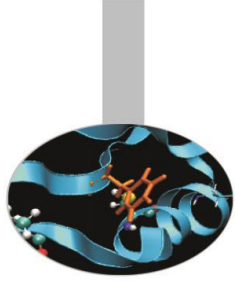
- The GNU suite does not provide exactly equivalent options.
  - The best option is to specify: **`-fdump-tree-all`**
  - which prints out alot of stuff (but not exactly in user-friendly form).

Optimising Loops



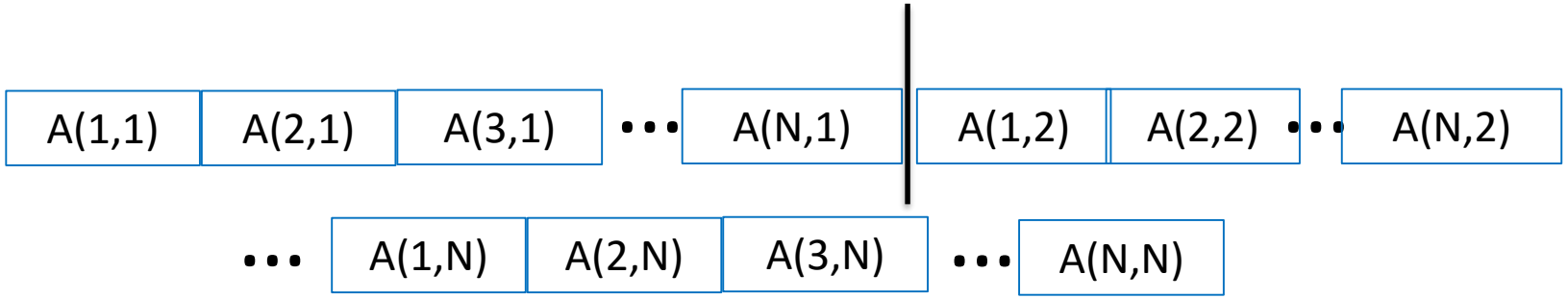
- Many HPC programs consume resources in loops where there are array accesses.
- Since main memory accesses are expensive principle goal when optimising loops is to maximise *data locality* so that the cache can be used. Another goal is to aid *vectorisation*.
- For simple loops the compiler can do this but sometimes it needs help.
- *Important to remember differences between FORTRAN and C for array storage*.
- But should always test the performance. For small arrays, in particular, the various optimisations may give worse results.

Reminder -Storing arrays in C and Fortran



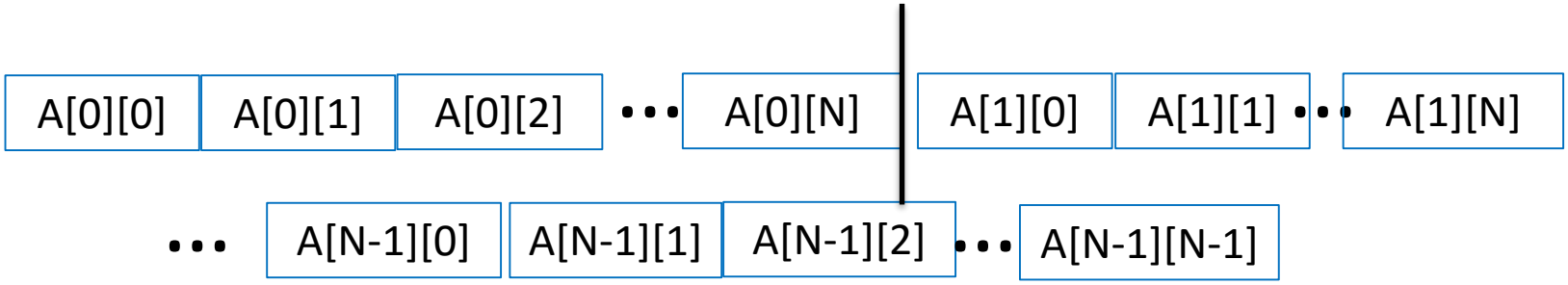
Integer, dimension :: A(N,N)

COLUMN-MAJOR

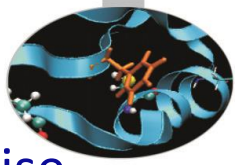


int A[N][N];

ROW-MAJOR



Loop optimisations

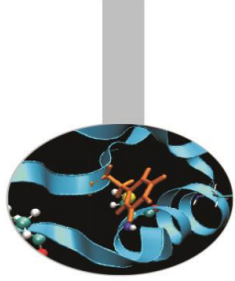


- First rule: always use the correct types for loop indices. Otherwise the compiler will have to perform real to integer conversions.
- FORTRAN compilers may indicate an error or warning, but usually tolerated

```
real :: i,j,k
....
do j=1,n
 do k=1,n
    do i=1,n
       c(i,j)=c(i,j)+a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

Compilation	integer	real
PLX gfortran –O0	9.96	8.37
PLX gfortran –O3	0.75	2.63
PLX ifort –O0	6.72	8.28
PLX ifort –O3	0.33	1.74
Plx pgif90	4.73	4.85
Plx pgif90 -fast	0.68	2.3
Fermi bglxlf –O3	64.78	104.1
Fermi bgxlf –O3	0.64	12.38

Loop optimisations: index reordering

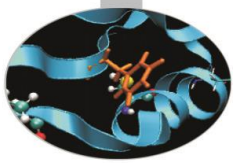


For simple loops, the compiler optimises well

```
do i=1,n
do j=1,n
do k=1,n
    c(i,j) = c(i,j) + a(i,k)*b(k,j)
end do
end do
end do
```

Compilation	J-k-i	i-k-j
Ifort –O0	6.72	21.8
Ifort –fast	0.34	0.33

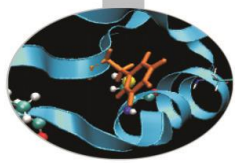
Practical session 1 – array indices



- Have a look at the matrix multiplication routines in the mult-matrix subdirectory.
- Have a look at the three variants, which should be the fastest?
- Compile with –O0 to prevent the compiler optimising the execution and time the executions.

```
ifort –O0 mult_matrix.f90
time ./mult_matrix
```

Loop optimisations – index reordering

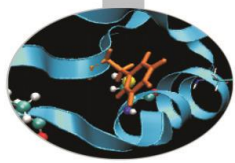


- For more complex, nested loops optimised performances may differ.
- Important to understand the cache mechanism!

```
do jj = 1, n, step
  do kk = 1, n, step
    do ii = 1, n, step
      do j = jj, jj+step-1
        do k = kk, kk+step-1
          do i = ii, ii+step-1
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

Compilation	j-k-i	i-k-j
(PLX) ifort -O0	10	11.5
(PLX) ifort -fast	1.	2.4

Loop optimisations -cache blocking



If the a,b,c, arrays fit into the cache, performance is fast

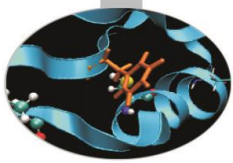
```
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1) {
    r = 0;
    for (k = 0; k < N; k = k+1){
          r = r + y[i][k]*z[k][j];
    }
   x[i][j] = r;
};
```

If not then performance is slow. By adding loops, can reduce data held such that it fits into cache.

```
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i = i+1)
      for (j = jj; j < min(jj+B-1,N); j = j+1)  {
        r = 0;
        for (k = kk; k < min(kk+B-1,N); k = k+1) {
            r = r + y[i][k]*z[k][j];
        }
        x[i][j] = x[i][j] + r;
      };
```

B=blocking factor

Loop optimisations – unrolling (or unwinding)

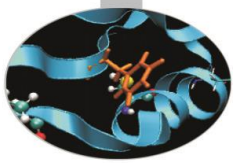


- Aim to reduce loop overhead (e.g. loop control instructions) by reducing iterations. Can also reduce memory accesses, and aid vectorisation.
- Can be done by replicating the code inside the loop.
- Most effective when the computations in the loop can be simulated by the compiler (e.g. stepping sequentially through an array) . Clearly, the no. of iterations should be known before execution.

```
for(int
i=0;i<1000;i++)
   a[i] = b[i] + c[i];
```

in some cases can eliminate a loop altogether

```
for(int i=0;i<1000;i+=4) {
     a[i] = b[i] + c[i];
     a[i+1] = b[i+1] + c[i+1];
     a[i+2] = b[i+2] + c[i+2];
     a[i+3] = b[i+3] + c[i+3];

}
```



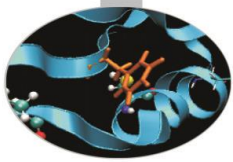
Loop optimisations – loop fusion

- A loop transformation which replaces multiple loops with a single one (to avoid loop overheads and aid cache use).
- Possible when two loops iterate over the same range and do not reference each other's data. (unless "loop peeling" is used)
- Doesn't always improve performance – sometimes cache is better used in two loops (*Loop fission*)

```
/* Unoptimized */
for (i = 0; i < N; i = i + 1)
   for (j = 0; j < N; j = j + 1)
     a[i][j] = 2 * b[i][j];

for (i = 0; i < N; i = i + 1)
   for (j = 0; j < N; j = j + 1)
     c[i][j] = K*b[i][j]+ d[i][j]/2
```

```
/* Optimized */
for (i = 0; i < N; i = i + 1)
  for (j = 0; j < N; j = j + 1)
    a[i][j] = 2 * b[i][j];
    c[i][j] = K*b[i][j]+d[i][j]/2
```

Loop optimisations - fission



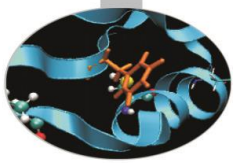
- The opposite of Loop fusion, i.e. splitting a single loop into multiple loops.
- Often used when:
  1. computations in single loop become too many(which can lead to “register spills”).
  2. If the loop contains a conditional: create 2 loops, one without conditional for vectorisation.
  3. Improve memory locality.

```
for (j=0; j<n; j++) {
   for (i=0; i<n; i++) {
      b[i][j] = a[i][j];
   }
   for (i=0; i<n; i++) {
      c[i][j] = b[i+m][j];
   }
}
```

```
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      b[i][j] = a[i][j];
   }
}
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      c[i][j] = b[i+m][j];
   }
}
```

non local access

local access



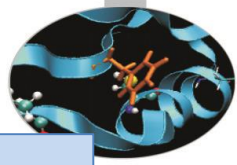
Array of Structures (AoS) vs Structure of Arrays (SoA)

- Depends on access patterns, but for vectorised C/C++ usually preferable to have SoA rather than AoS since array elements are contiguous in memory.
- SoA also usually uses less memory because of data alignment.

```
// AoS
struct node {
   float x,y,z;
// other data
};

struct node NODES[N];
```

```
// SoA
struct node {
   float x[N];
   float y[N];
   float z[N];
//other data
};
struct node NODES;
```

Example



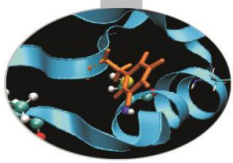
```
// Array of structures
struct node {
   float x,y,z;
   int n;
};

struct node NODES[N];

for (i=0;i<N;i++) {
   NODES[i].x=1;
   NODES[i].y=1;
   NODES[i].z=1;
}
for (i=0; i<N; i++) {
   x=NODES[i].x;
   y=NODES[i].y;
   z=NODES[i].z;
   sum+=sqrtf(x*x+y*y+z*z);
```

```
// Struct of Arrays
struct node {
   float x[N];

struct node NODES;

for (i=0;i<N;i++) {
   NODES.x[i]=1;
   NODES.y[i]=1;
   NODES.z[i]=1;
}
for (i=0; i<N; i++) {
   x=NODES.x[i];
   y=NODES.y[i];
```

```
icc -O2 -opt-report 2 -o soa soa.c -lm
soa.c(22:1-22:1):VEC:main:  LOOP WAS VECTORIZED
soa.c(29:1-29:1):VEC:main:  LOOP WAS VECTORIZED
```

```
icc -O2 -opt-report 2 -o aos aos.c -lm
aos.c(18:1-18:1):VEC:main:  loop was not vectorized: not inner loop
aos.c(19:4-19:4):VEC:main:  loop was not vectorized: low trip count
aos.c(25:1-25:1):VEC:main:  LOOP WAS VECTORIZED
```

Practical session 2 – loop optimisations

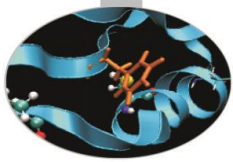


In the optimisations/loops directory you will find some examples of different types of loops. We suggest:

- Compile the example programs, switching off for the moment the automatic vectorisation to see what non-vector optimisations are performed by the Intel compiler.
- Check the optimisations performed with the –qopt-report option of icc/ifort.
- Try also varying the amount of information the compiler has by removing explicit definition of the array sizes (e.g. loop4.c)

```
module load intel
icc -qopt-report -no-vec -c loop.c
less loop.optrpt
```

Vectorisation

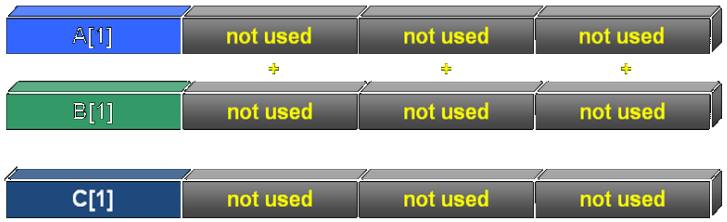


- Modern processors have dedicated circuits and SIMD instructions for operating on blocks of data (“vectors”) rather than single data items.

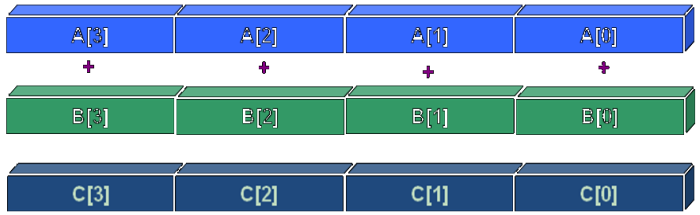
```
c(0) = a(0) + b(0)
c(1) = a(1) + b(1)
c(2) = a(2) + b(2)
c(3) = a(3) + b(3)
```

non vectorised

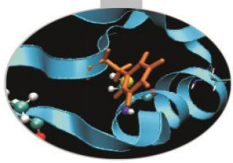
e.g. 3 x 32-bit unused integers



vectorised

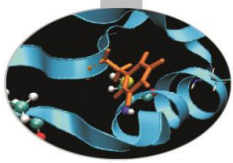


Vectorisation evolution



- SSE: 128 bit registers (intel Core - AMD Opteron)
  - 4 floating/integer operations in single precision
  - 2 floating/integer operations in double precision
- AVX: 256 bit registers (intel Sandy Bridge - AMD Bulldozer)
  - 8 floating/integer operations in single precision
  - 4 floating/integer operations in double precision
- MIC: 512 bit registers (Intel Knights Corner - 2013)
  - 16 floating/integer operations in single precision
  - 8 floating/integer operations in double precision

Vectorisation



- Loop vectorisation can increase dramatically the performance.
- But to be vectorisable a loop must obey certain criteria, in *particular the absence of dependencies* between separate iterations.
- Other criteria include:
  - Countable (constant number of iterations)
  - Single entry/exit points (no branches, unless implemented as masks)
  - Only the internal loop of a nested loop
  - No function calls (unless inlined or using a vector version of the function)
- Note that AVX can give different numerical results (e.g. Fused Multiply Addition)

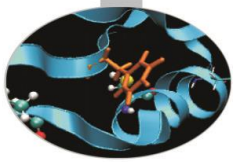
Vectorisation can be difficult..

One of the following code snippets vectorises, the other one doesn't

```
subroutine vec

integer, parameter ::n=1000
integer :: i
real :: a(n),b(n),c(n)

do i=2,n
   a(i-1)=a(i)+1
enddo

end subroutine
```

```
subroutine vec

integer, parameter ::n=1000
integer :: i
real :: a(n),b(n),c(n)

do i=2,n
   a(i) = a(i-1) + 1
end do

end subroutine
```

Vectorisation Algorithms

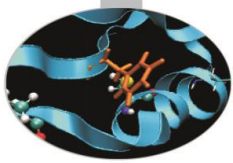


- Different algorithms performing the same task can behave differently wrt vectorisation.
  - Gauss-Seidel: dependency between iterations, not vectorisable.

```
for( i = 1; i < n-1; ++i )
   for( j = 1; j < m-1; ++j )
      a[i][j] = w0 * a[i][j] +
      w1*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
```

```
for( i = 1; i < n-1; ++i )
   for( j = 1; j < m-1; ++j )
     b[i][j] = w0*a[i][j] +
     w1*(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]);
for( i = 1; i < n-1; ++i )
   for( j = 1; j < m-1; ++j )
      a[i][j] = b[i][j];
```

Helping the vectoriser



- Some “coding tricks” can block vectorisation:
  - vectorisable

```
for( i = 0; i < n-1; ++i ){
        b[i] = a[i] + a[i+1];
}
```

  - non vectorisable because x is needed for the next iteration.

```
x = a[0];
for( i = 0; i < n-1; ++i ){
   y = a[i+1];
   b[i] = x + y;
   x = y;
}
```

- If the code hasn’t vectorised then you can help the compiler by:
  - modifying the code to make it vectorisable
  - inserting compiler directives to force the vectorisation

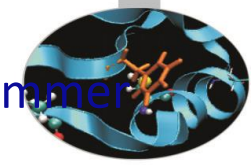
Helping the vectoriser

```
for (i=0; i<N;i++) {
      a[i]=b[i]+c[i];
      a[i+1]=a[i]+2*d[i]
}
```

```
for (i=0; i<N;i++) {
      temp[i]=b[i]+c[i];
      a[i+1]=temp[i]+2*d[i]
;
      a[i]=temp[i];
}
```

Remove output dependency with a new temporary variable.

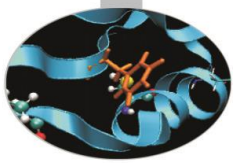
Helping the vectoriser



- If the programmer knows that a dependency indicated by the programmer is only apparent then the vectorisation can be forced with compiler-dependent directives.
  - Intel FOTRAN: !DIR$ simd
  - Intel C:#pragma simd
- so if we know that inow ≠ inew then there is in fact no dependency

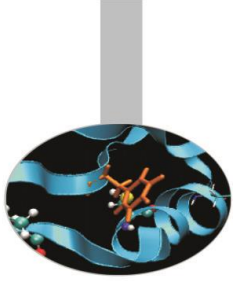
```
do k = 1,n

!DIR$ simd
   do i = 1,l
   ...
    x02 = a02(i-1,k+1,inow)
    x04 = a04(i-1,k-1,inow)
    x05 = a05(i-1,k ,inow)
    x06 = a06(i, k-1, inow)
    x19 = a19(i ,k ,inow)

    rho =+x02+x04+x05+x06+x11+x13+x14+x15+x19
    a05(i,k,inew) = x05 - omega*(x05-e05) + force
    a06(i,k,inew) = x06 - omega*(x06-e06)
```

Inlining



- A manual or compiler optimisation which replaces a call to the function with the body of the function itself.
  - eliminates the cost of the function call and can improve instruction cache performance
  - makes inter-procedure optimisation easier
- In C/C++ the keyword **`inline`** is a “suggestion”
- Not every function is “inlineable” – depends on the compiler.
- Can cause increase in code size, particularly for large functions.
- Intel: **`-inline=n`** (0=disable, 1=keyword, 2=compiler decides)
- GNU: **`-finline-functions`**, **`-finline-limit=n`**
- In some compilers activated at high optimisation levels

Common Subexpression Elimination (CSE)



- Sometimes identical expressions are calculated more than once. When this happens may be useful to replace them with a variable holding the value.
- This

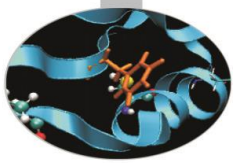
```
A = B+C+D
E = B+F+C
```

  requires 4 sums. But the following

```
A =(B+C) +D
E =(B+C) +D
```

  requires 3 sums.
- Careful: the floating point result may not be identical
- Another use is to replace an array element with a scalar to avoid multiple array lookups.

CSE and function calls



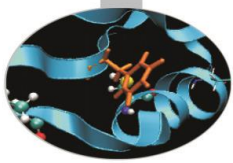
- By altering the order of the calls the compiler doesn't know if the result is affected (possible side-effects)
- 5 function calls, 5 products

```
x=r*sin(a)*cos(b);
y=r*sin(a)*sin(b);
z=r*cos(a);
```

- 4 function calls, 4 products (1 temporary variable)

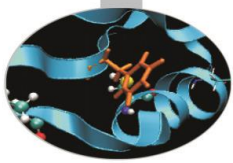
```
temp=r*sin(a)
x=temp*cos(b);
y=temp*sin(b);
z=r*cos(a);
```

CSE: Limitations



- Loops which are too big:
  - The compiler works with limited window sizes: it may not detect which quantity to re-use
- Functions:
  - If I change the order of the functions do I still get the same result?
- Order and evaluations:
  - Only at high levels of optimisation does the compiler change the order of operations (usually –O3 and above).
  - In some expressions it is possible to inhibit the mechanism with parantheses (the programmer is always right!).
- Since intermediate values are used will increase use of registers (risk of “register spilling”).

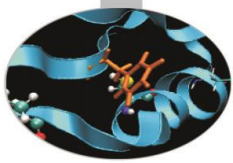
Static and dynamic allocation



- Static allocation in principle can help the compiler optimise by providing more information. But
  - the code becomes more rigid
  - in parallel computing dynamic allocation is very useful

```
integer :: n
parameter(n=1024)
real a(1:n,1:n)
real b(1:n,1:n)
real c(1:n,1:n)
```

```
real, allocatable, dimension(:,:) :: a
real, allocatable, dimension(:,:) :: b
real, allocatable, dimension(:,:) :: c
print*,'Enter matrix size'
read(*,*) n
allocate(a(n,n),b(n,n),c(n,n))
```



Static and Dynamic Allocation

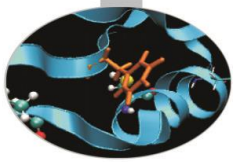
- For recent compilers, performances are often similar for static and dynamic allocations.
  - e.g. matrix-matrix multiplication

Compiler	Static	Dynamic
PLX ifort –O0	6.72	18.26
PLX ifort –fast	0.34	0.35

- Note that static allocations use the “stack”, which is generally limited.
- In the bash shell you can use the ulimit command to see and (possibly) set the stack.

```
ulimit –a
ulimit –s unlimited
```

Dynamic allocation in C



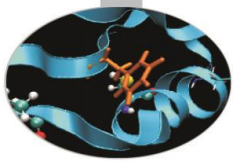
- C doesn't have a native 2-d array (unlike FORTRAN) but instead uses arrays of arrays.
- Static allocation guarantees all the values are contiguous in memory

```
double A[nrows][ncols];
```

- Dynamic allocation can be inefficient, if not done carefully

```
/* Inefficient array allocation */
/* Allocate a double matrix with many malloc */
double** allocate_matrix(int nrows, int ncols) {
   double **A;
   /* Allocate space for row pointers */
   A = (double**) malloc(nrows*sizeof(double*) );
   /* Allocate space for each row */
   for (int ii=1; ii<nrows; ++ii) {
      A[ii] = (double*) malloc(ncols*sizeof(double));
   }
return A;
}
```

Dynamic array allocation in C/2



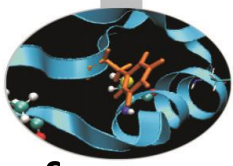
- Better to allocate a linear (1D array) and use it as matrix (*index linearisation*).

```
/* Allocate a double matrix with one malloc */
double* allocate_matrix_as_array(int nrows, int ncols) {
double *arr_A;
/* Allocate enough raw space */
arr_A = (double*) malloc(nrows*ncols*sizeof(double));
return arr_A;
}
..
arr_A[i+ncols+j]
```

- If necessary can add a matrix of pointers pointing to the allocated array

```
/* Allocate a double matrix with one malloc */
double** allocate_matrix(int nrows, int ncols, double* arr_A) {
double **A;
/* Prepare pointers for each matrix row */
A = new double*[nrows];
/* Initialize the pointers */
for (int ii=0; ii<nrows; ++ii) {
A[ii] = &(arr_A[ii*ncols]);
}
return A;
}
```

Aliasing and restrict

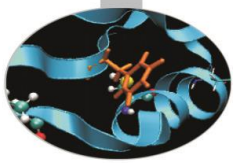


- In C *aliasing* occurs if two pointers point to the same area of memory.
- Aliasing can severely limit compiler optimisations:
  - difficult to invert the order of the operations, particularly if passed to a function
- The C99 standard introduced the **`restrict`** keyword to indicate that aliasing is not possible:

```
void saxpy(int n, float a, float *x, float* restrict y)
```

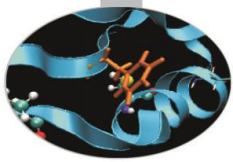
- In C++ it is assumed that aliasing cannot occur between pointers to different types (strict aliasing).

Aliasing and Restrict /2



- FORTRAN assumes that the arguments of a procedure cannot point to the same area of memory
  - except for arrays where in any case the indices allow a correct behaviour
  - or for pointers which are used anyway as arguments
  - one reason why FORTRAN optimises better than C!
- It is possible to configure the aliasing options at compile time
  - GNU (solo strict-aliasing): **`-fstrict-aliasing`**
  - Intel (complete elimination): **`-fno-alias`**
  - IBM (no overlap per array): **`-qalias=noaryovrlp`**

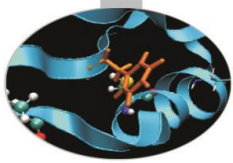
Practical session 3 - vectorisation



- Repeat practical session 2 but this time switch on the vectorisation.
- Some hints
  - For testing vectorisation use subroutines or functions rather than whole programs. The optimiser may decide not to use vectorisation if it knows the problem size is small.
  - If testing on KNL, remember to use the KNL vectorisation flag:

```
ifort –xMIC-AVX512 –c myprog.f90
```

Optimizing parallel programs



- A complicated subject which needs to be done in conjunction with profiling and tracing tools.
- After profiling, two major starting points for optimizing parallel programs:
  1. Modifying the program to reduce synchronisations and overlap calculation with communication as much as possible
  2. Match the parallel processes (MPI, OpenMP) to the underlying hardware.
- In the next few slides we will look only at point 2.

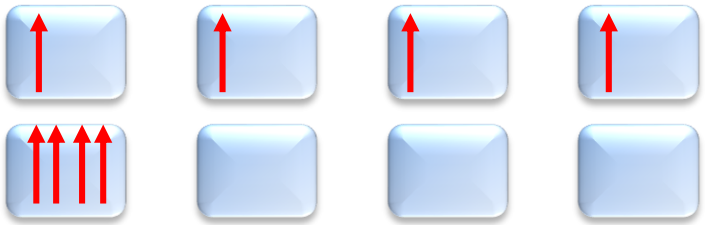
Thread and task affinity (pinning)

With the number of tasks or threads/node increasing it is important to know to which cores they have been assigned. This is true for all NUMA HPC systems but particularly relevant for, e.g. KNL, with it high number of cores. The performance difference can be significant !

A complicated topic because the ***pinning*** is controlled by a variety of options or environment options, many of which are intel-specific and liable to change.

For KNL, see for example,

https://software.intel.com/en-us/articles/process-and-thread-affinity-for-intel-xeon-phi-processors-x200



KMP_AFFINITY=scatter

KMP_AFFINITY=compact

Thread and task pinning – some variables

Variable	Example	Description
I_MPI_DEBUG=[0-5]	export I_MPI_DEBUG=5	Shows the logical cores owned by each MPI rank (the affinity). Default affinity=scatter
KMP_AFFINITY=[scatter ,compact, proclist={..}]	export KMP_AFFINITY=compact, verbose.	Changes the affinity to, e.g compact. The verbose option shows the result of the change.
KMP_PLACE_THREADS, KMP_HW_SUBSET=<t> T (new)	export KMP_HW_SUBSET=2T	threads per core (by default all 4 thds/core are used)
OMP_NUM_THREADS= n	export OMP_NUM_THREADS=4	For an OpenMP program total number of threads, for hybrid threads/MPI rank.
OMP_PLACES=[cores,th reads]	export OMP_PLACES=0,1,2,3,4,…271,272	Specifies hardware resource. Used with OMP_PROC_BIND
OMP_PROC_BIND=[clo se, spread]	export OMP_PROC_BIND=spread,close	How OpenMP threads are bound to resources

Difficult to use correctly. Better use KMP_HW_SUBSET

Pinning MPI tasks

The MPI task pinning can be queried by setting

I_MPI_DEBUG=5

For NUMA-aware MPI (e.g. Intel) the default affinity should be “sensible”, e.g. the ranks should be spread evenly between the sockets in a node.

Finer control (e.g at the KNL tile level) can be controlled with variables such as I_MPI_PIN_PROCESSOR_LIST or I_MPI_PIN_DOMAIN.

For multi-nodes, the –perhost option indicates how many tasks/knl.

```
$ export I_MPI_DEBUG=5
$ mpirun -np 2 ./simple
…
[0] MPI startup(): Rank    Pid      Node name
Pin cpu
[0] MPI startup(): 0       698092   r086c15s03
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18
30,31,32,33,68,69,70,71,72,73,74,75,76,77,78,79,
80,81,82,83,84,85,86,87,88,89,90
221,222,223,224,225,226,227,228,229,230,231,232,
233,234,235,236,237}
[0] MPI startup(): 1       698093   r086c15s03
{34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49
,50,51,52,53,54,55,56,57,58,59,60
,61,62,63,64,65,66,67,102,103,104,105,106,107,10
8,109,110,…,265,266,267,268,269,270,271}
```

KNL example (68 physical cores, 272 virtual cores)

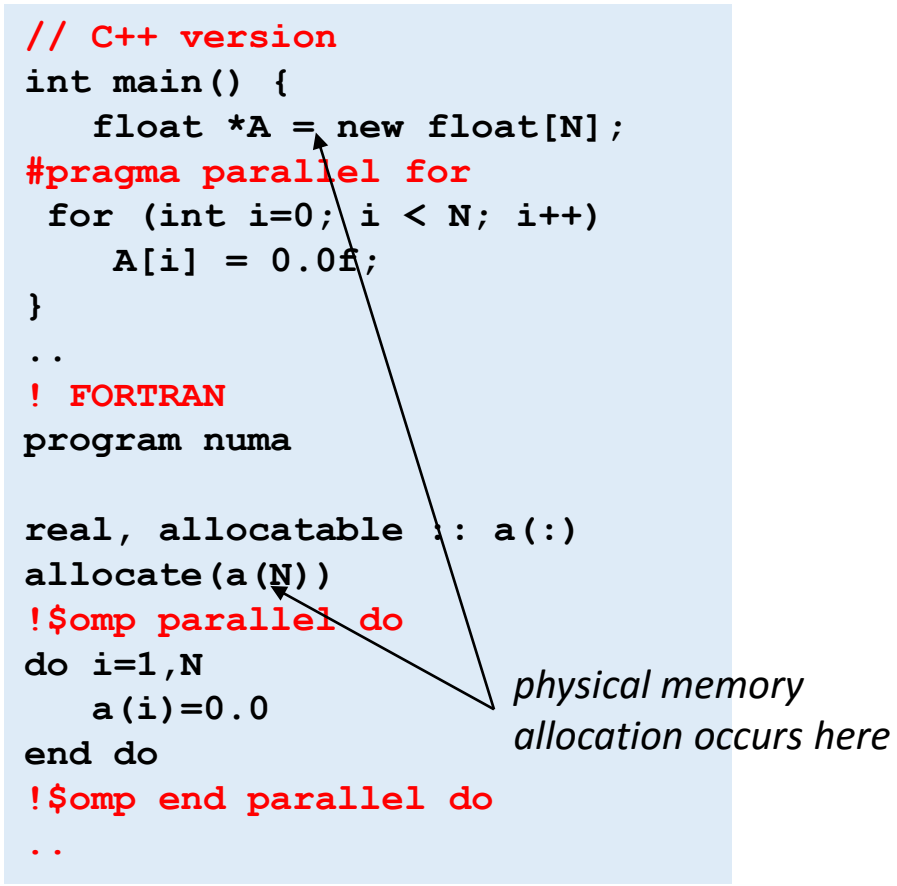
Binding memory to NUMA nodes

Once the thread or task has been pinned to a sub-NUMA cluster it makes sense also to bind the memory objects.

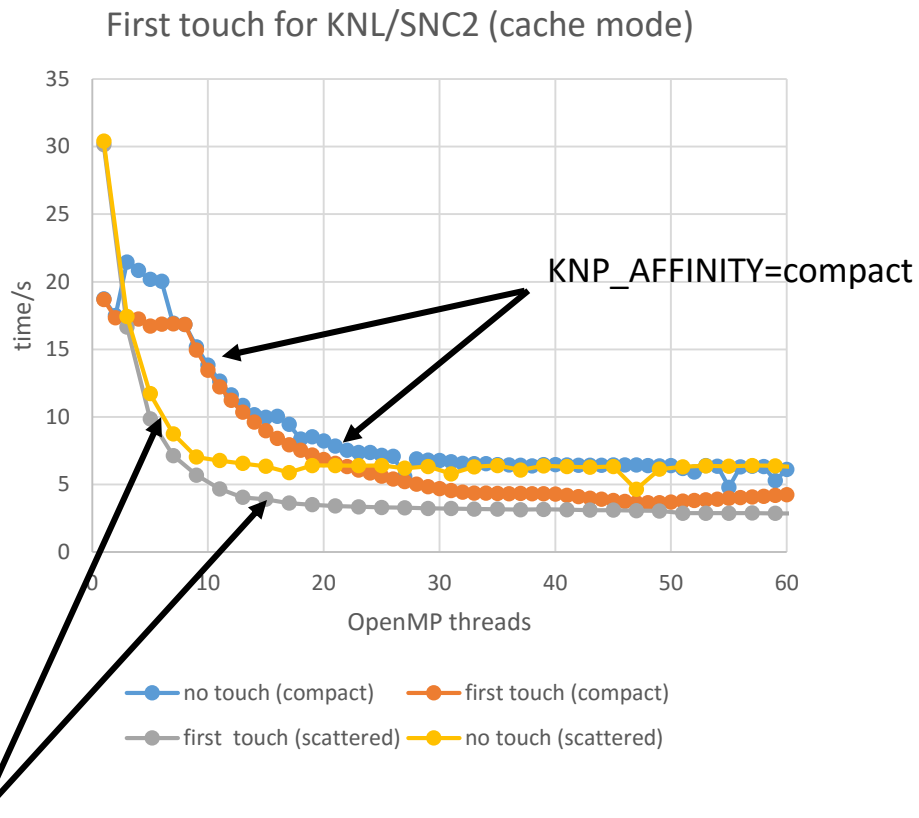
As with any NUMA device the default allocation policy in Linux is "**first touch**", when a thread first writes into a newly allocated array (touches it) the memory page is allocated on the thread's NUMA node.

For this reason also array initialisation should be done in the parallel region: otherwise the array is allocated on the numa mode of the master thread. [assuming the shared array is large]

```
// C++ version
int main() {
   float *A = new float[N];
#pragma parallel for
 for (int i=0; i < N; i++)
    A[i] = 0.0f;
}
..
! FORTRAN
program numa

real, allocatable :: a(:)
allocate(a(N))
!$omp parallel do
do i=1,N
   a(i)=0.0
end do
!$omp end parallel do
..
```

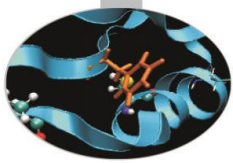


Binding memory to numa nodes - Example



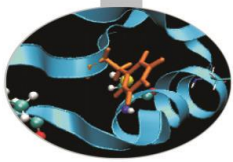

```
integer :: n=1000000000
allocate(a(n),b(n),c(n))
!!$omp parallel do
      do i=1,n
         a(i)=10.0
         b(i)=2.0
         c(i)=1.0
      enddo
!!$omp end parallel do
      t=mysecond()
   do j=1,10
!$OMP parallel do
      do i=1,n
         a(i)=b(i)+d*c(i)
      enddo
!$OMP end parallel do
```

Input/Output



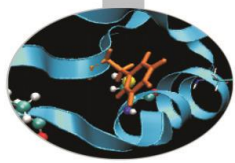
- I/O is performed by the operating system and:
  - results in a system call
  - empties the pipeline
  - destroys the coherence of data in the cache
  - is very slow
- Rule 1: Do not mix intensive computing with I/O
- Rule 2: read/write data in blocks, not a few bytes at a time (the optimum block size depends on filesystem)

Fortran I/O examples



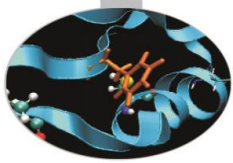
```
do k=1,n ; do j=1,n ; do i=1,n
write(69,*) a(i,j,k)                              ! formattated
enddo ; enddo ; enddo
do k=1,n ; do j=1,n ; do i=1,n
write(69) a(i,j,k)                                ! binary
enddo ; enddo ; enddo
do k=1,n ; do j=1,n
write(69) (a(i,j,k),i=1,n)                        ! columns
enddo ; enddo
do k=1,n
write(69) ((a(i,j,k),i=1),n,j=1,n)                ! matrices
enddo
write(69) (((a(i,j,k),i=1,n),j=1,n),k=1,n)        ! block
write(69) a                                       ! dump
```

FORTRAN I/O performances



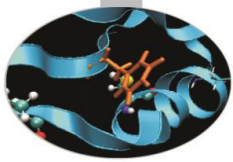
Option	Seconds	Kbytes
Formatted	81.6	419430
Binary	81.1	419430
Columns	60.1	268435
Matrix	0.66	134742
Block	0.94	134219
Dump	0.66	134217

I/O Summary



- Reading/writing formatted data is slow.
- Better to read/write binary data.
- Read/write in blocks.
- Choose the most efficient filesystem available.
- Note that although writing is generally buffered, the impact on performance can be significant.
- For parallel programs:
  - avoid having every task perform read/writes
  - use instead MPI I/O, NetCDF or HDF5, etc.

Summary



- Most programmers optimise their codes by simply increasing the optimisation level during the compilation but with complex programs the compiler normally needs help.
- Many serial optimisations, regardless of language (C, Fortran,..), work towards optimal cache and vector performance – particularly essential for hybrid HPC architectures (e.g. GPU, KNL).
- Parallel optimisation is an advanced topic, greatly assisted by profilers and performance tools. With many core technologies such as KNL important to take into account MPI task and OpenMP thread affinities.