Adventures with automatic code transformations
Chris Maynard – Met Office
16/9/2015
New Met Office HPC

Phase 1A 2x 4 cabinets (now)
  20k cores (Haswell)
  + 3PB lustre (each) + 6PB
  ~ IBM power7 machine

Phase 1B 2x 13+ cabinets (Q1 2016)
  100k cores (Broadwell)
  ~ 6X IBM power7 machine

Phase 1C Q1 2017
  New IT hall
  5.5 MVA facility (upgradeable)
  30+ cabinets + 6PB lustre
  ~250k cores (Skylake)

£97M investment
  – Cray XC40
Optimisations – Global model

Accumulated effect of lots of small(ish) changes IO (Lustre), Comms, quality of Fortran, compiler opts run-time env

Open MP threads quality and coverage affinity: bound to real cores

A. Voysey

IBM Power7
Cray XC40 (unoptimised)
Cray XC40 (unoptimised w/ gcr tol)
Cray XC40 (optimised)
Cray XC40 (optimised w/ gcr tol)
Cray XC40 (unoptimised w/ gcr tol, no -cc numa_node)
Cray XC40 (optimised w/ gcr tol, no -cc numa_node)
UM performance

Operational 25\textsuperscript{th} August (IBM switch off 17\textsuperscript{th} September)
Intel Xeon 2.6 GHz (Haswell) 16-core dual socket

N768 ~ 17km res.
150M grid-points (operational)
N1024 ~ 12km res.
267M grid-points
MPI + 2 OMP threads
Cray compiler, IO server
Different solver config

P. Selwood, A. Malcolm
UM ENDGame Helmholtz Solver: Data Layout

Performance dominated by preconditioner `tri_sor`

Data layout is lexicographical \((i,j,k)\) in UM

Loop order is red-black in `tri_sor` (threading)

BAD cache use and SIMD

Change to linear red and black data layout

Update from last year: New hardware and compiler (`ifort15`)

New variants tried

Compare Xeon Phi (STFC DL) with Xeon (ib Archer and hsw MO)
Code Transformations: Toy Code

RB: declaration + ifort directive for assume_aligned

RB gen: replace array bounds and loop counters with literal constants

RB: loop counters

RB dbl gen: replace

RB dbl: block loop on veclen (fortran parameter)
RB use directives
!dir$
assume_aligned x:64
and compiler option
-align array64byte
gen: \rightarrow generated.
array sizes and loop bounds replaced
with literal constants (numbers)
dbl: \rightarrow loops blocked
with vec len. 8 for DP KnC

Serial stand alone solver compiled with
-mmic -simd -O3 -openmp
Fully populated node

One replica per core: no comms
Volume dependence

\(nx \times ny\) 70 levels
Fully populated, 4 threads per core
Full UM with roughly \(16^2\) fits in 8Gbytes of memory

Small volumes lcg fits into cache
Larger volumes RB is faster
SIMD works better with compiler directives and literal constants
Ivybridge (Archer: Cray XC30)

1 OMP thread per core
work per thread constant
RB helps for CPU (better cache usage)
No SIMD benefit for literal constants
Excellent (weak) scaling

CPU single socket
60 KnC c.f. 12 Ivb cores
96*96*60=192*240*12
No Hyper-threads

- lcg
- RB var
- RB gen
- RB dbl var
- RB dbl gen
Haswell (MO: Cray XC40)

CPU single socket
60 KnC c.f. 16 Haswell cores
96*96*60=192*180*16
No Hyperthreads

1 OMP Thread per core
Work per Thread constant
RB helps for CPU
No SIMD benefit for literal constants
Cray compiler
Excellent (weak) scaling
Architecture comparison

Socket comparison
large vol, 96² on *Phi*
- Data parallel 4 Threads per Core

RB helps on all arch
RB dbl no extra benefit
Good threading perf.
Code gen for SIMD *Phi*
- No benefit on CPU
Compiler, Cray similar to Intel
UM Summary

Data layout match Loop order is key (this is not news)
Boost performance on all architectures
**Red-black** data layout unlocks KNC threading perf
SIMD performance is poor
Auto-gen: Replace array declarations and loop counters with literal constants \(x2\) speed up
examine output from `ifort -opt-report`
inner loops are SIMD-ised, but peel and remainder loops literal constants \(\rightarrow\) no peel and remainder
`!$OMP SIMD` doesn’t help
Very large volume `tri_sor()` perf on KNC beats HSW!
Implementing in full model: - data re-order: + overlap comms and compute
Gung Ho
Fortran – high level language
Abstraction of the numerical mathematics
Implementation and architecture is hidden
Code – text which conforms to the semantics and syntax of the language definition
Compiler transforms code into machine code for specific processors

This abstraction is **broken** by parallel/performance/memory features exposed
Hacked back together with MPI, Open MP, Open ACC, OpenCL, CUDA, PGAS, SIMD, compiler directives
Libraries, languages (extensions), directives and compiler (specific) directives
Programming model II

Desired API

Actual API
Domain Specific Language (DSL)
Abstraction of the numerical mathematics of our domain
Implementation and architecture is hidden
Code – text which conforms to the semantics and syntax of the language definition
*Transform text into standard language*
Compiler transforms code into machine code for specific processors

We don’t yet know what language Gung Ho will be written in, but it will be Fortran!

F2K3 + OO style
Plus some extra rules
These should actually help the natural scientist
Layered architecture - PSyKAI

Single model
- PSy
- Algs
- kernel

Driver
- read partition
- initialisation
- finalise

Infrastructure
- e.g. halo_exchange
Some important computational consequences of the mathematics

Unstructured horizontal mesh - indirect memory addressing

regular (graded) vertical mesh – contiguous in memory
Cells, columns and dofmaps

2+1 d mesh – column of cells, stacked (and distorted)

do cell = 1, ncells
  
  do k = 0, nlevels -1
    do df = 1, ndf
      data(map(df,cell) + k) = ...
    end do
  end do
end do
Algorithm layer

High level view of problem
Written in terms of global fields and operations on fields (kernels)
Written by NS in Fortran + PSyKAI API rules

! Dynamo DSL F2K3 OO code

\[
\text{call invoke( calc_exner_kernel_type(exner, rho, theta, chi, qr), & axpy_type(dt, rho_inc, rho_n, rho) )}
\]

! Transformed code is F2K3 calling to PSy later
! recompute latest exner value
\[
\text{call invoke_calc_exner_kernel( exner, rho, theta, chi, qr )}
\]
! alpha \times x + y: alpha is scalar, x and y are fields
\[
\text{call invoke_axpy(dt, rho_inc, rho_n, rho)}
\]
! recompute latest exner value

! or based on data access descriptors and comms patterns (in kernel header)
! code generator could decide to fuse these calls together
\[
\text{call invoke_calc_exner_plus_axpy(exner, rho, theta, chi, qt, dt, rho_inc, rho_n)}
\]
Kernel metadata
How many fields
Data access descriptors
PSyClone can figure out comms pattern
Which function space
FEM basis funcs
(Not required for FD)
Natural Scientist writes metadata
PSyClone reads and generates kernel stub
NS writes rest of kernel
Algorithm layer and Kernel layer conform to PSyKAI API. PSyKAI API is the Domain Specific Language. Then a Python code (called PSyClone) can:
1. Parse the Fortran code
2. Decide what and when communications are required
3. Decide when shared parallelism can be exploited
4. Generate appropriate Fortran Code for PSy Layer
   1. Includes the calls the kernels
Complex parallel code is hidden from Natural Scientist.
OpenMP shared memory parallelism

```
! Look-up colour map
!
call a_proxy*vspace*get_colours(ncolour, ncp_colour, cmap)
!
do colour=1,ncolour
   !$omp parallel do default(shared), private(cell,map_w2,map_w3,map_w0), schedule(static)
   do cell=1,ncp_colour(colour)
      !
      map_w2 => a_proxy*vspace*get_cell_dofmap(cmap(colour, cell))
      map_w3 => b_proxy*vspace*get_cell_dofmap(cmap(colour, cell))
      map_w0 => c_proxy*vspace*get_cell_dofmap(cmap(colour, cell))
      !
      call ru_code(nlayers, a_proxy%data, b_proxy%data, c_proxy%data, d_proxy(1)%data, &
                  d_proxy(2)%data, d_proxy(3)%data, ndf_w2, undf_w2, map_w2, basis_w2, &
                  diff_basis_w2, boundary_dofs_w2, ndf_w3, undf_w3, map_w3, basis_w3, ndf_w0, &
                  undf_w0, map_w0, basis_w0, diff_basis_w0, nqp_h, nqp_v, wh, wv)
   end do
   !$omp end parallel do
end do
```

Relatively simple logic, but need to order (colour) cells to avoid race condition when incrementing dofs on shared entities
Does it work?

Single Socket Haswell (Exclusive)
400x16x16 60 timesteps

OMP scaling single node
Parallelism

Dynamo shows good scaling for OMP threads across shared memory
MPI for distributed memory is coming soon
Developing OpenACC version to target GPU
Conference in November – FEM order, element, vertical, transport still be decided

Science code in Algorithm and Kernel layers was unchanged

Science code in Algorithm and Kernel layers was unharmed
Open ACC

Psy layer hand written code, offload a kernel, (column)
Logic similar to openmp control of memory with data region
1. manage data transfer
2. only simple types (acc2.0)

need to annotate kernel source match procedure declaration
SIMD (vector/warp) level parallelism
PSyClone doesn’t (yet) have this capability
In general, we will need to produce performance optimised kernels, beyond current event horizon
The Free Lunch is over
Exploiting parallelism for performance is:
  a) difficult
  b) work (lots thereof, re-writing code)
(Semi) automatic code transformations are a good way to achieve performance portability

If everyone writes own DSL, fragments effort, reduced ability to leverage other software
Trying to solve a hardware problem in software
Need a programming model
International exascale software project didn’t get anywhere