The challenge of code modernization for the Exascale: methodology and early experiments

Eric Petit, eric.petit@prism.uvsq.fr
Aristote Seminar, Feb. 5th 2015, X, France

The COncurrency and LOcality Challenge

EXascale Algorithms and Advanced Computational Techniques
Increasing number of nodes, cores, accelerators

– Some **resources do not scale**
  
  • Memory per core, Bandwidth, Coherence protocol, Network interconnect, Fault tolerance

– **Multiplication of hierarchical levels** => **Non uniformity and Heterogeneity**
  
  • Frontier are becoming fuzzier => Distributed/shared? Software/hardware? "Core" definition? Compute capabilities, imbalance…
  
  • Different scales: BW, memory size, performance
  
  • Global events: barrier, broadcast, memory coherency
**Evolutions are requested** for applications, runtimes and programming models
Introduction to the Exascale programming challenge

More concurrency
- Enough independent tasks
- Communication overlap
- Privatize memory to avoid communication (& sync)
- Remember Amdahl: the more core, the higher the proportion of the sequential code is

More locality
- Memory
  • Core level, Socket level (including HWA), Network level
- But also communication
  • Synchronization, Data

We need both for performance scalability
Why not experimenting in the original application?

- Full applications are complex and costly to execute at scale
  - Difficulty to experiment ground breaking solutions
  - Cost of the experiments (time, PY, CPUs)
  - Need proof of concept demonstrating ROI to decide

- Codes and use-cases might not be easily shared with the community

- Need a strong and daily support of the application developer

- Portability of the solution
  - Over specialization
  - Learning curve, even in the same company/context
The Proto-App Concept

Aka mini-app, proxy-app (NERSC trinity, Argonne CESAR, the Montevo project…)

**Objectives**: Reproduce at scale the behavior of a set of HPC applications and support the development of optimizations that can be translated into the original applications
- Easier to execute, modify and re-implement

If you cannot make the application open-source, you can at least open-source the problems.
- Support community engagement
- Reproducible and comparable results
- Interface with application developers
Two alternatives with pros and cons

- **Build-up** (upcoming mini-FMM, stay tune)
  - ‘Mini-app’ that mimic a full application with simpler physic
  - All aspects are explored
  - No/Less IP issue(s)
  - No specific problem targeted
  - Behavior at scale?
  - Representativeness?
  - Feedback to the real code?
  - Use cases?

- **Strip down** (mini-FEM)
  - ‘Proxy-app’ which extracts and refines a particular kernel from an application
  - Target a specific issue
  - Must be representative at scale
  - Easy feedback to the user
  - Only a part of the application is addressed
  - Problem coupling?
  - Use cases generation?
  - IP (code and use case)

IMHO I prefer the second one, building multiple proto-apps from an application to expose the different problems => however it requires the application developer and end-user experience
• CSR matrix assembly from an **unstructured mesh**
  
• Proto-application extracted from DEFMESH (Dassault Aviation)

• Successfully ported back into AETHER (CFD code at Dassault Aviation)

Reduction done on each edge from all neighboring elements

Edges update (+= reduction) must be sequential

\[
\begin{align*}
X_{ij} & \neq 0 \text{ if there is an edge between } i \text{ and } j \\
(X) \text{ Sparse and symmetric matrix}
\end{align*}
\]
Mini-FEM DC: a Scalable Nested Parallelism for **Unstructured** Meshes

- Current parallelization approaches
- Will not be efficient on future
- 1000’s nodes of 1000 cores
- Exascale nodes!
  \[\Rightarrow\] Efficient hybrid parallelization is requested

---

**Efficient on current architectures**

**Sub-optimal on future architectures**

**Data duplications**

**Synchronisations**

---

**Simple to implement**

- Bad locality (can be mitigated using blocking)
- High memory bandwidth requirements
- Global synchronizations
D&C: a Scalable Nested Parallelism for Unstructured Meshes

Can create many independent tasks

=> **Concurrency**

Leaves data set can be downsized at will to fit into caches

=> **Data locality**

Only one synchronization per task between neighbors

=> **Sync locality**

Only Log (N) sync on the critical path

=> **Sequential part minimization**

```
function compute (partition)
  if Node is not a leaf
    spawn compute (partition.left)
    compute (partition.right)
    sync
    compute (partition.sep)
  else
    FEM_assembly (partition)
  end
```

- Open source DC_lib (LGPL)
- Open source proto-application
- Can be reuse in place for any loop over elements or loop over nodes in FEM codes
Measuring Locality

(~Similar to Nested dissection for fill-minimization in sparse LU, see in MUMPS)
Measuring Locality
Coloring at node or socket level has proven to be a bad idea, however…

Coloring has been designed in the context of vector machines

A core itself is a vector machine…

=> Let’s try coloring!

The following results use the vectorization model as described in our PPOPP 2015 paper:

for each element E
    myColor = 0, mask = 1
    for each neighbor elements NE
        neighborColor |= elemToColor[NE]
    while (neighborColor & mask)
        neighborColor = neighborColor >> 1
        myColor++
    elemToColor[E] = (mask << myColor)
SOA Longest Color Strategy

Subdomain Max Size: 50
Vector Length: SSE

Vec Ratio 69.6%

Subdomain Max Size: 50
Vector Length: AVX

Vec Ratio 20.9%

Subdomain Max Size: 50
Vector Length: AVX512

Vec Ratio 0.1%

Subdomain Max Size: 200
Vector Length: SSE

Vec Ratio 90.9%

Subdomain Max Size: 200
Vector Length: AVX

Vec Ratio 73.8%

Subdomain Max Size: 200
Vector Length: AVX512

Vec Ratio 43.8%

Subdomain Max Size: 500
Vector Length: SSE

Vec Ratio 94.9%

Subdomain Max Size: 500
Vector Length: AVX

Vec Ratio 85.4%

Subdomain Max Size: 500
Vector Length: AVX512

Vec Ratio 66.3%
Poor vectorization ratio

Probably not enough data parallelism in the data that fit in cache…

However the small amount of available data parallelism is badly exploited: heuristics for large domains are not efficient on smaller domains fitting into cache!

– Longest colors constraint the number of colors

=> We do not need such a constraint, we want ‘long enough’ colors only!
for each element E
myColor = 0, mask = 1
for each neighbor elements NE
    neighborColor |= elemToColor[NE]
while (neighborColor & mask) |
    colorCard[myColor] >= VEC_SIZE)
    neighborColor = neighborColor >> 1
myColor++
    elemToColor[E] = (mask << myColor)
colorCard[myColor]++
Bounded Color Strategy

Bounded => ~10% improvement on the partition size fitting into cache.
Vectorization implementation

Sequential loop of vectors, no need for a parallel loop
⇒ Permutation allows to forget about the colors,
⇒ Align the data dependencies on iteration frontier
⇒ Just remember offset for the next vector size.
=> Future work: mix vector size using mask/padding

for each color C in a leaf
vec_for elem in [0:C_SIZE%VEC_SIZE]
seq_for elem in [C_SIZE%VEC_SIZE:C_SIZE]

Without reordering
vec_for elem in [0:offset]
seq_for elem in [offset:LEAF_SIZE]

With reordering

10% shorter with bounded
With increasing vector size we need increasing dataset size to be efficient on unstructured data
- But cache size per core is decreasing
- And vector size is getting larger
=> Can’t run efficiently in L1 with vectors on current phi !!!
(and L2 on phi 😞…)

The current gather operations require large compute intensity to be overlapped
⇒ Some loops are faster not being vectorized
Implementation trade-off

Data Locality

Pure D&C
Structured
Hybrid
Pure coloring

Data Unstructuredness
Data Parallelism
## Table 1. Vectorization expected speed-ups for a leaf size of 200.

<table>
<thead>
<tr>
<th>vecSize</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>8 (native)</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded vecRatio</td>
<td>0.96</td>
<td>0.90</td>
<td>0.83</td>
<td>0.76</td>
<td>0.55</td>
<td>0.02</td>
</tr>
<tr>
<td>Expected SU</td>
<td>1.27</td>
<td>1.36</td>
<td>1.38</td>
<td>1.37</td>
<td>1.27</td>
<td>1.01</td>
</tr>
<tr>
<td>Longest vecRatio</td>
<td>0.91</td>
<td>0.74</td>
<td>0.44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected SU</td>
<td>1.25</td>
<td>1.32</td>
<td>1.20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
– Best HW vector size is application dependent
– The choice of the architect is a tradeoff based on benchmarks
  ⇒ Co-design is required
  ⇒ Provide him with your proto-apps!
– Larger/faster memory
  • Not the actual trend, at least not smaller and slower would be good
  • However the bandwidth is increasing, but not all the algorithms can benefit from it. (e.g. Massive SPMD model like in GPU programming)
Yes it already existed in the past

Long vector machines are back…

Actually it is more accurate to say: high ratio vector length/memory machines are back

Predicates, masks, complex vector operations, divergence, N1/2…

New branch, taking a new direction from a solid basis of previous work => we are not doomed!
Some Mini-FEM proto-app results

- Ideal scaling
- D&C Hybrid (Cilk)
- D&C (Cilk)
- Ref Improved (MPI)
- Ref (MPI)
- Coloring (Cilk)

Speed-Up vs Number of Cores

- Hyper-threading

Speed-Up vs Number of Cores
OpenMP vs. Cilk on Phi

New OpenMP version of DC_lib:
- Significant difference on the physical cores
  - Larger overhead of the OpenMP runtime?
- Hyper-threads compensate on larger core counts…
- However not really promising for the future
- Tests on BlueGene coming soon!
OpenMP vs. Cilk on 4 Phi

The OpenMP version of DC_lib

![Graph showing parallel efficiency and speed-up for different numbers of cores for OpenMP and Cilk.](image)

- Ideal scaling
- D&C (Cilk)
- D&C (OpenMP)
As for future work

- In exa2ct and coloc, all our developments are open source
  - Coria Yales2 for load balancing of chemistry and lagrangian particles (exa2ct)
  - More experiments on the proto-app of the multigrid solver of DLR Tau provided by Tsystem (exa2ct)
  - Experimenting GASPI RMA async one sided and compare to MPI3.0 in distributed DC version of Mini-FEM + solver (Coloc)
  - FMM with async one sided, efficient data placement and load balancing, and efficient shared memory parallelization (many-core requirement) (Coloc)
Other requests and ideas are welcome! ;)

Questions?