Speeding up Scientific Codes in HPC Architectures by Code Modernization: Lessons Learned (1/2)

José M. García
jmgarcia@um.es

Parallel Computing Architecture Group (GACOP)
University of Murcia
Murcia (Spain)

Academic Training Lecture Programme @ CERN
Geneve (Switzerland), June 2019
The importance of speeding-up programs\(^1\):

"There's Plenty of Room at the Top," Leiserson, et. al., to appear

\(^1\)"A new Golden Age for Computer Architecture", J. Hennessy and D. Patterson in the 2018 ACM A.M. Turing Award Lecture
**Motivation**

A first important fact

- A rewriting the code in **C** from **Python** —a typical high-level, dynamically typed language— increases performance **47-fold**
- Using **parallel loops** running on many cores yields a factor of approximately **7**
- Optimizing the **memory layout** to exploit caches yields a factor of **20**
A final factor of 9 comes from using the hardware extensions for doing single instruction multiple data (SIMD) parallelism operations that are able to perform 16 32-bit operations per instruction

⇒ the final highly optimized version runs more than 62,000x faster
**Motivation**

**A second important fact**

---

**The International Exascale Challenge**

- **Goal**: Build a High-Performance Computer (*Supercomputer*) that achieves a Peak Performance of 1 ExaFlops ($10^{18}$ FLOPs) under the 20 MWatts envelope.

- Exascale computing will not just allow present solutions to run faster, but will enable new solutions not affordable with today’s HPC technology.

- Exascale computing means real capability improvement in science and engineering.

- Exascale computing will enable breakthrough science.

- Two broad types of applications: *simulations* (modelling) and *big data*.
Motivation
The International Exascale Challenge
taken from Yelick’s talks
**Motivation**

The International Exascale Challenge

**Worldwide Exascale Roadmaps**

- From K computer... to Post K with domestic technology.
- From Tianhe-2... to Tianhe-2A with domestic technology.
- From the PPP for HPC to future PRACE systems... with domestic technology?

José M. García

Code Modernization

CERN - June 2019
**Motivation**

**The European Exascale Approach**

---

**The EuroHPC Joint Undertaking**

- **Development**
  - In **2017**, seven European countries signed the European declaration on High-Performance Computing.
  - In **October 2018**, the EU, together with 24 EU member states and Norway, established the European High Performance Computing Joint Undertaking (EuroHPC JU), a public-private partnership.

- **Objectives**
  - Its mission will be to develop, deploy, extend, and maintain in the EU an integrated world-class supercomputing and data infrastructure capable of at least $10^{18}$ calculations per second (so-called exascale computers).
  - The **goal** is to have a **exascale supercomputer based on European technology** in the global **top 3** supercomputers by **2022**.
  - In addition, EuroHPC JU will develop and support a highly competitive and innovative HPC ecosystem.
MOTIVATION
THE INTERNATIONAL EXASCALE CHALLENGE

THE EUROHPC JOINT UNDERTAKING

- Challenges
  - Develop a European microprocessor and European exascale systems
  - Develop exascale software and applications
  - Widen use of HPC and address the HPC-related skills gap

- The EuroHPC JU Ramp-Up Phase (2019–2020)
  - The EuroHPC JU will acquire and install two top-five pre-exascale machines and several mid-range supercomputers by 2020
  - The EuroHPC JU will invest €1.4 billion in the period 2019-2020

This is a European project of the size of Airbus in the 1990s and of Galileo in the 2000s
In addition to build a physical exascale computer, there are also some...

**Exascale Application Development Challenges**

- Adopting new mathematical approaches
- Algorithmic or model improvements
- **Porting** to multicore and accelerator-based architectures
- Exposing and **optimizing** additional parallelism
- Leveraging optimized libraries
**Intel High-end Architectures**

- **Portability**: Run x86 code
- **General-purpose** processors
- Latency-oriented vs. throughput-oriented processors
- Flexibility & *Cheaper* cost per unit
Intel high-end architectures @ The Top500 list

From the last list: November 2018
The research of our group has been always centered around the topic of Supercomputing

MAIN RESEARCH LINES

- 2000 - 2012: Parallel computer architecture: Multiprocessor on chip (CMPs)
- 2008 - until now: Code modernization: For GPUs and CMPs architectures
- 2015 - until now: HPC for Deep Neural Networks

MEMBER OF NOES

- European Network of Excellence on "High Performance and Embedded Architecture and Compilation" (HiPEAC)
- European Network of Excellence on "Transnational Access Programme for a Pan-European Network of HPC Research Infrastructures and Laboratories for scientific computing" (HPC-EUROPE)
- HyperTransport Technology Consortium
- Spanish E-Science Network
The aim of this talk is to guide (in)experienced software developers to optimize important applications for Intel high-end architectures.

- Identify potential problems and bottlenecks, solutions and trade-offs
- Modernization code process: Best practices for a single node

- Metrics: Performance (wall-clock time), speedup and parallel efficiency
- Intel high-end architectures: Portability and broad range
Motivation

Structure of the Talk

- Background
  - Parallelism: Technology trends and parallel programming
  - Intel high-end architectures (multicores & manycores)
  - Code modernization: Best practices

- Practical examples
  - Stencil codes: Scientific apps that operate over an N-dimensional data structure that changes over time, given a fixed computational pattern
  - Semantic Web: A Semantic dataset generator that transforms relational or XML data into semantic repositories
  - Ant Colony Optimization (ACO): A Bio-inspired metaheuristic applied to a wide range of NP-hard combinatorial optimization problems

- Conclusions and Lessons learned

- Future lines: Domain-Specific Languages (DSLs) and Domain-Specific Architectures (DSAs)
Motivation

Structure of the talk

- Background
  - Parallelism: Technology trends and parallel programming
  - Intel high-end architectures (multicores & manycores)
  - Code modernization: Best practices

- Practical examples
  - Stencil codes: Scientific apps that operate over an N-dimensional data structure that changes over time, given a fixed computational pattern
  - Semantic Web: A Semantic dataset generator that transforms relational or XML data into semantic repositories
  - Ant Colony Optimization (ACO): A Bio-inspired metaheuristic applied to a wide range of NP-hard combinatorial optimization problems

- Conclusions and Lessons learned

- Future lines: Domain-Specific Languages (DSLs) and Domain-Specific Architectures (DSAs)
Motivation

Structure of the talk

- Background
  - Parallelism: Technology trends and parallel programming
  - Intel high-end architectures (multicores & manycores)
  - Code modernization: Best practices

- Practical examples
  - Stencil codes: Scientific apps that operate over an N-dimensional data structure that changes over time, given a fixed computational pattern
  - Semantic Web: A Semantic dataset generator that transforms relational or XML data into semantic repositories
  - Ant Colony Optimization (ACO): A Bio-inspired metaheuristic applied to a wide range of NP-hard combinatorial optimization problems

- Conclusions and Lessons learned

- Future lines: Domain-Specific Languages (DSLs) and Domain-Specific Architectures (DSAs)
OUTLINE

1 BACKGROUND

2 CASE STUDY: 3-DStencil Codes

3 CASE STUDY: SemanticWeb and Bioinformatics

4 CASE STUDY: ACO

5 Conclusions and Future Research Lines
BACKGROUND
Technology trends & Parallel Programming

TECHNOLOGY EVOLUTION

42 Years of Microprocessor Trend Data

- Transistors (thousands)
- Single-Thread Performance (SpecINT x 10^3)
- Frequency (MHz)
- Typical Power (Watts)
- Number of Logical Cores

Year


from karlrupp.net
**BACKGROUND**

**TECHNOLOGY TRENDS & PARALLEL PROGRAMMING**

**UNIPROCESSOR PERFORMANCE (SINGLE CORE)**

Performance = highest SPECInt by year; from Hennessy & Patterson [2018]
BACKGROUND
Technology trends & Parallel Programming

FACTS
- Transistor use (dark silicon) was affected by
  - Power wall
  - Memory wall
- Clock speed was affected by
  - Power wall
- Automatic instruction parallelism was affected by
  - ILP (*Instruction-Level Parallelism*) wall
  - Memory wall
- Speculation & Out-of-order execution was affected by
  - Power wall
  - Memory wall
BACKGROUND
Technology trends & Parallel Programming

Solutions

- Exploiting specialization: Accelerators
- Exploiting data parallelism (SIMD): Using wider vector instructions
- Exploiting thread parallelism (TLP): Using multi-cores

Hardware keeps evolving and Software must catch up!
A general multicore view

- Vector parallelism
- Thread parallelism
- Shared memory (L3 cache and RAM)

From Colfax HowTo Slides
Exploiting data parallelism (SIMD): Using vector instructions

Scalar Instructions

\[
\begin{array}{ccc}
4 & + & 1 = 5 \\
0 & + & 3 = 3 \\
-2 & + & 8 = 6 \\
9 & + & -7 = 2 \\
\end{array}
\]

Vector Instructions

\[
\begin{array}{ccc}
4 & 1 & 5 \\
0 & 3 & 3 \\
-2 & 8 & 6 \\
9 & -7 & 2 \\
\end{array}
\]

from Colfax HowTo slides

The **wider** the SIMD registers the **better** performance achieved using vectorization
Exploiting thread parallelism (TLP)

Threads are streams of instructions that share memory address space

The higher the core number the better performance achieved using thread parallelization

from Colfax HowTo slides
We have used the OpenMP framework in this work

**OpenMP Main Features**

- **OpenMP** = “Open Multi-Processing” = computing-oriented framework for shared-memory programming ([https://www.openmp.org/](https://www.openmp.org/))

  - This is *de facto* parallel programming standard
  - The current version is OpenMP 5.0 (Nov 2018)
  - OpenMP covers the *entire hardware spectrum* from embedded and accelerator devices to high-end multicore systems with shared-memory
  - The core elements of OpenMP are the constructs (mainly *pragmas*) for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables.

  - **Thread creation examples:** `#pragma omp parallel for` and `#pragma omp task`
**BACKGROUND**

**TECHNOLOGY TRENDS & PARALLEL PROGRAMMING**

**Using Vectors: Two Approaches**

**Automatic Vectorization** →

```c
double A[vec_width], B[vec_width];

for(int i = 0; i < vec_width; i++)
    A[i] += B[i];
```

← **Explicit Vectorization**

```c
double A[8], B[8];
__m512d A_v = _mm512_load_pd(A);
__m512d B_v = _mm512_load_pd(B);
A_v = _mm512_add_pd(A_v, B_v);
_mm512_store_pd(A, A_v);
```

from Colfax HowTo slides

**Helping automatic vectorization**

- **icc compiler pragmas**: `#pragma ivdep` or `#pragma vector`
- OpenMP has also **pragmas** for helping automatic vectorization (e.g. `#pragma omp simd`)
**BACKGROUND**

**TECHNOLOGY TRENDS & PARALLEL PROGRAMMING**

**PRINCIPLES OF PARALLEL COMPUTING**

- Granularity – how big should each parallel task be
- Locality – moving data costs more than arithmetic
- Load balance – don’t want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Performance modeling/debugging/tuning

- Finding enough parallelism (Amdahl’s Law)

All of these things make parallel programming even harder than sequential programming
Motivation
Background
Stencil
Web
ACO
Conclusions & Future
Parallel
Intel_HPC_Arch
Code_Mod.

**BACKGROUND**

**INTEL HIGH-PERFORMANCE ARCHITECTURES**

Evolution of vector parallelism on Intel Architectures

Source: Intel Developer Zone
Evolution of thread parallelism on Intel high-end Architectures

<table>
<thead>
<tr>
<th></th>
<th>5100</th>
<th>5500</th>
<th>5600</th>
<th>E5</th>
<th>IVB</th>
<th>HSW</th>
<th>BDW</th>
<th>SKL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td>Threads</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>36</td>
<td>44</td>
</tr>
</tbody>
</table>

**Source:** Intel Developer Zone

**IVB:** Ivy Bridge  
**BDW:** Broadwell  
**HSW:** Haswell  
**SKL:** Skylake  
**KNC:** Knights Corner  
**KNL:** Knights Landing
**BACKGROUND**

**Intel High-Performance Architectures**

**Xeon family**
- General-purpose: Suitable for any workload
- 1-, 2-, 4-sockets: NUMA architecture
- Highly parallel
- Resource-rich
- Forgiving performance: High single-thread performance

**Xeon Phi (1st Gen - Knights Corner or KNC)**
- Accelerator/coprocessor x86 based
- 61 in-order cores @ low frequency (1.2Ghz)
- Low single-thread performance
- High Memory Bandwidth: 320 GB/sec
- Custom operating system on board

**Xeon Phi (2nd Gen - Knights Landing or KNL)**
- Improved single-thread performance (3x vs. KNC)
- 36 tiles interconnected by 2D mesh
- Processor fully binary compatible with Xeon line
BACKGROUND
Intel High-Performance Architectures

Intel Xeon Phi Knights Landing: Major novelties

**KNL: On-Package High-Bandwidth Memory**
- 16 GB of MCDRAM (Multi-channel, i.e., high bandwidth memory)
- 5x bandwidth vs. DDR4
- 5x power efficiency vs. DDR4
- 3 operating modes: Cache, Flat and Hybrid

**KNL: Clustering Modes**
- For applications sensitive to cache traffic (latency-bound)
- Three different modes:
  - None: all-to-all
  - As an SMP architecture: quadrant/hemisphere
  - As a NUMA architecture: SNC-4/SNC-2
**BACKGROUND**

**OUR EVALUATION TEST BED**

Evaluation environment: two Intel Xeon multicore and two Intel Xeon Phi manycore (KNC and KNL)

<table>
<thead>
<tr>
<th></th>
<th>Xeon v2</th>
<th>Xeon v4</th>
<th>Xeon Phi KNC</th>
<th>Xeon Phi KNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microarchitecture</td>
<td>Ivy-Bridge</td>
<td>Broadwell</td>
<td>MIC</td>
<td>MIC</td>
</tr>
<tr>
<td>Sockets</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>2.6 GHz</td>
<td>2.2 GHz</td>
<td>1.238 GHz</td>
<td>1.4 GHz</td>
</tr>
<tr>
<td>Cores/socket</td>
<td>8 out-of-order</td>
<td>20 out-of-order</td>
<td>61 in-order</td>
<td>68 out-of-order</td>
</tr>
<tr>
<td>Threads/core</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>VPU Width</td>
<td>256 bits (AVX)</td>
<td>256 bits (AVX-2)</td>
<td>512 bits (AVX-512)</td>
<td>512 bits (AVX-512)</td>
</tr>
<tr>
<td>Peak Performance (SP)</td>
<td>665.6 GFLOPs</td>
<td>1408 GFLOPs</td>
<td>2020 GFLOPs</td>
<td>6092 GFLOPs</td>
</tr>
<tr>
<td>L1d-cache size/core</td>
<td>32 KB</td>
<td>32 KB</td>
<td>32 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td>L2-cache size/core</td>
<td>256 KB</td>
<td>256 KB</td>
<td>512 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>L2-cache size (total)</td>
<td>4 MB</td>
<td>10 MB</td>
<td>30.5 MB</td>
<td>34 MB</td>
</tr>
<tr>
<td>L3-cache</td>
<td>20 MB</td>
<td>50 MB</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>DRAM size</td>
<td>32 GB</td>
<td>128 GB</td>
<td>16 GB</td>
<td>192 GB</td>
</tr>
<tr>
<td>Peak Memory Bandwidth</td>
<td>59.7 GB/s</td>
<td>76.8 GB/s</td>
<td>320 GB/s</td>
<td>76.8 GB/s</td>
</tr>
<tr>
<td>MCDRAM size</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>16 GB</td>
</tr>
<tr>
<td>MCDRAM Bandwidth</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>400 GB/s</td>
</tr>
</tbody>
</table>
**BACKGROUND**

Our evaluation test bed

Evaluation environment: two Intel Xeon multicore and two Intel Xeon Phi manycore (KNC and KNL)

<table>
<thead>
<tr>
<th></th>
<th>Xeon v2</th>
<th>Xeon v4</th>
<th>Xeon Phi KNC</th>
<th>Xeon Phi KNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microarchitecture</td>
<td>Ivy-Bridge</td>
<td>Broadwell</td>
<td>MIC</td>
<td>MIC</td>
</tr>
<tr>
<td>Sockets</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>2.6 GHz</td>
<td>2.2 GHz</td>
<td>1.238 GHz</td>
<td>1.4 GHz</td>
</tr>
<tr>
<td>Cores/socket</td>
<td>8 out-of-order</td>
<td>20 out-of-order</td>
<td>61 in-order</td>
<td>68 out-of-order</td>
</tr>
<tr>
<td>Threads/core</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>VPU Width</td>
<td>256 bits (AVX)</td>
<td>256 bits (AVX-2)</td>
<td>512 bits (AVX-512)</td>
<td>512 bits (AVX-512)</td>
</tr>
<tr>
<td>Peak Performance (SP)</td>
<td>665.6 GFLOPs</td>
<td>1408 GFLOPs</td>
<td>2020 GFLOPs</td>
<td>6092 GFLOPs</td>
</tr>
<tr>
<td>L1d-cache size/core</td>
<td>32 KB</td>
<td>32 KB</td>
<td>32 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td>L2-cache size/core</td>
<td>256 KB</td>
<td>256 KB</td>
<td>512 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>L2-cache size (total)</td>
<td>4 MB</td>
<td>10 MB</td>
<td>30.5 MB</td>
<td>34 MB</td>
</tr>
<tr>
<td>L3-cache</td>
<td>20 MB</td>
<td>50 MB</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>DRAM size</td>
<td>32 GB</td>
<td>128 GB</td>
<td>16 GB</td>
<td>192 GB</td>
</tr>
<tr>
<td>Peak Memory Bandwidth</td>
<td>59.7 GB/s</td>
<td>76.8 GB/s</td>
<td>320 GB/s</td>
<td>76.8 GB/s</td>
</tr>
<tr>
<td>MCDRAM size</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>16 GB</td>
</tr>
<tr>
<td>MCDRAM Bandwidth</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>400 GB/s</td>
</tr>
</tbody>
</table>
Intel High-Performance Architectures

**Intel Tools**

- Use Intel Parallel Studio to develop HPC code
  - icc & icpc compilers, vectorization & parallelization reports
- Other useful Intel profiling tools
  - Use Intel VTune Amplifier to find hotspots
  - Use Intel Advisor to get hints on how to enhance vectorization
  - Use Intel Inspector to find and debug data races
**Background**

**Code Modernization**

---

**Bringing Codes Into the Parallel Age**

- **Code modernization** can mean many things, from using a modern language to optimizing performance.
- Code modernization tries to **extract the maximum performance from an application** and take full advantage of **modern hardware**.
- Just upgrading to new hardware does not always result in better application performance. It may take modifications to the code to reap those performance gains.
- There is no “one recipe, one solution” technique.
- Much of Intel’s code modernization work comes out of its global network of Intel Parallel Computing Centers.
**Optimization areas: Node and cluster level**

<table>
<thead>
<tr>
<th>Node-level</th>
<th>Cluster-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar tuning</td>
<td>Communication</td>
</tr>
<tr>
<td>Vectorization</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Parallelization</td>
<td></td>
</tr>
</tbody>
</table>
**Background**

**Code modernization: Best practices**

### Sequential Code

- **Competitive code** in high-level languages (latest versions of C, C++, or Fortran)
- **Scalar Tuning**: Strength reduction, precision control, and other compiler-friendly practices

### Vectorization

- Programmers have to facilitate the compiler’s task by **rearranging** the source code
  - Unit-stride access: AoS (Array of Structures) to SoA (Structure of Arrays)
  - Data alignment
  - Container padding and eliminate peel loops
  - Eliminate multiversioning
- Programmers have to facilitate the compiler’s task by **adding some hints** into the source code

---

José M. García  
Code Modernization  
CERN - June 2019
**BACKGROUND**

**CODE MODERNIZATION: BEST PRACTICES**

### Thread Parallelism
- From sequential to parallel: data-based or task-based parallelization
- Exposing more parallelism: loop collapse and strip-mining
- Minimizing load imbalanced: scheduling
- Minimizing synchronization
- Avoiding false sharing
- Thread affinity
- NUMA data locality

### Memory Layout
- Exploit memory access: data locality, bandwidth memory tuning
- Optimize data re-use in caches
- Loop tiling: cache blocking and unroll-and-jam
- Loop fusion: re-use data as soon as possible
- Loop permutation: achieve unit-stride access
OUTLINE

1 BACKGROUND

2 Case Study: 3-D Stencil Codes

3 Case Study: Semantic Web and Bioinformatics

4 Case Study: ACO

5 Conclusions and Future Research Lines
The problem

- 3-D Stencil codes are iterative kernels which updates data elements according to some fixed predetermined pattern
- 3D stencil codes involve access to large volumes of data and suffer from poor cache performance because data reuse is minimal
- The goal is to evaluate the behaviour of these codes on Xeon Phi KNC, and improve their execution time over a naïve parallel code running in Xeon v2

Code Modernization features

- Evaluate automatic vectorization (512-bit vector wide)
- Analyze thread parallelism: tuning its parameters
- Exploit memory locality: Cache blocking
- Target platforms: Xeon v2 (16 cores, 2 threads/core), KNC (61 cores, 4 threads/core), KNL (68 cores, 4 threads/core)
Partial differential equations (PDEs) are the core of many problems. Usually solved by the finite-difference method, which gives an approximate solution in an iterative way. The solution is computed by updating each of the input elements with correctly weighted values of neighboring elements. This computing pattern is known as Stencil.

Stencils depend on:
- The number of spatial dimensions (1-D, 2-D, 3-D, etc)
- The number of neighbors in each dimension (1, 2, 3, 4, ...)
- The time order of the code: Element \( (Time_t) = F(Time_{t-1}, Time_{t-2}, \ldots) \)
**CASE STUDY: 3-D STENCIL CODES**

**BACKGROUND**

- Implemented as a time loop plus a triple nested loop along the entire data structure (3D)
- Computations are applied until meeting either a convergence criteria or a certain number of time steps

**ALGORITHM: GENERIC 3-D STENCIL SOLVER KERNEL**

```plaintext
1: for time = 0; time < TimeMax; time ++ do
2:     for z = 1; z < depth − BorderSize; z + + do
3:         for y = 1; y < height − BorderSize; y + + do
4:             for x = 1; x < width − BorderSize; x + + do
5:                 stencil_solver_kernel();
6:         end for
7:     end for
8: end for
9: tmp = Input_Grid; Input_Grid = Output_Grid; Output_Grid = tmp;
10: end for
```

* width, height, depth are the dimensions of the data set including border (halo) points.*
**Experimental methodology**

- Three 3-D Stencil kernels have been evaluated
  - **Acoustic** diffusion code: 7 point spatial with 2nd order in time
  - **Isotropic seismic** wave propagation code: 25 point spatial with 2nd order in time
  - **Heat** conduction code: 11 point spatial with 1st order in time
- A limit of 1,000 time-steps was set for the simulations of the three Stencil kernels (this sufficiently guaranteed the convergence of the problem).
- As recommended, we performed 2 executions of the stencil prior to running the 1000 iterations as "warm-up"
- Performance figures are given for **double-precision numbers**, and the execution times shown are the average of 10 independent runs
- Standard deviation is not shown, but was insignificant
- Unless we specify other thing, evaluations on Xeon Phi KNC and KNL have been carried out with **balanced affinity parameter**, and **compact** on Xeon v2 and v4.
- The default scheduling policy is **static**.
**SPECIFICATIONS OF SOFTWARE TEST BED**

- The OS for all the platforms is Linux CentOS (different versions on each platform)
- The KNC system also runs Intel MPSS 3.4.3
- Codes are built using Intel’s *icc* compiler with the optimization level `-O3`
- The option `-mmic` is set when compiling for Xeon Phi KNC.
CASE STUDY: 3-D STENCIL CODES

CODE MODERNIZATION: STARTING POINT

**BASE VERSION**

- A straightforward sequential implementation of the algorithm written in C/C++ for each of the Stencil kernel codes
- A naïve parallelization of these codes adding the `#pragma omp parallel for` before the nested triple loop that traverses the data input

**OPTIMIZED SCALAR VERSION**

- Applied scalar optimizations
  - Arithmetic operations strength reduction
  - Reorder of operations and data access
  - Use of the qualifier `const`
**Case Study: 3-D Stencil Codes**

**Code Modernization:** Evaluation of Base + Scalar Version

### Thread Scalability in Xeon v2 and Xeon Phi KNC

#### Xeon Phi KNC (1st gen.) vs. Xeon v2

61 cores vs. 16 cores

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acoustic</td>
<td>715</td>
<td>786</td>
<td>490</td>
<td>394</td>
<td>392</td>
<td>392</td>
</tr>
<tr>
<td>Seismic</td>
<td>2134</td>
<td>2494</td>
<td>1393</td>
<td>849</td>
<td>697</td>
<td>687</td>
</tr>
<tr>
<td>Heat</td>
<td>655</td>
<td>733</td>
<td>387</td>
<td>296</td>
<td>278</td>
<td>270</td>
</tr>
</tbody>
</table>

#### Xeon Phi KNC

<table>
<thead>
<tr>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>Acoustic</td>
</tr>
<tr>
<td>Seismic</td>
</tr>
<tr>
<td>Heat</td>
</tr>
</tbody>
</table>

*(time in secs.)*

- Scalability: Xeon v2 is limited to 16 threads and KNC to 128 threads
- Memory accesses are limiting scalability
- Poor benefit from using Xeon Phi KNC
Vectorization approach

- We have followed the **automatic** vectorization because of portability issues
- The vectorization report from the compiler showed that loops have not been vectorized

Therefore, we have **rearranged** the data layout and given some hints to ease the vectorization process:

**Data allocation:**
- We allocated **all rows** of the 3D arrays **consecutively** in memory (i.e., row major order)
- We mapped the **unit stride** dimension to the **inner loop** in nested loop iterations as it produced a better use of cache lines
- Then, the dataset was accessed in order of planes (layers), columns, and finally rows from outer to inner level
**Data alignment**: All Data structures started with an address aligned to 64
- We used `_mm_malloc(Data, 64)` instead of `malloc()`
- Additionally, we gave hints to the compiler as `__assume_aligned(Data, 64)` or `#pragma vector aligned`

**Align padding**: We padded the inner dimension of multi-dimensional arrays to guarantee alignment for each row of the matrix. The new width with padding was calculated as `width_PADD = (((width*sizeof(REAL))+63)/64)*(64/sizeof(REAL))`

**Data dependencies**: Finally, we put `#pragma ivdep` (or `#pragma omp simd`) before the loop for telling the compiler to ignore vector dependencies (which were false) and avoid loop multiversioning.
**Case Study: 3-D Stencil Codes**

**Code Modernization: Evaluation of Vectorization (I)**

Xeon Phi KNC (1st gen.) vs. Xeon v2
61 cores (4 th/core) vs. 16 cores (2 th/core)

---

**Results Obtained**

- The vectorized code on the KNC outperforms all kernels versions
- KNC shows a performance improvement close to 7X against Xeon v2
- Comparing with the baseline version running on KNC, around 4x.
**CASE STUDY: 3-D STENCIL CODES**

**CODE MODERNIZATION: EVALUATION OF VECTORIZATION (II)**

Xeon Phi KNL (2nd gen.) vs. Xeon Phi KNC (1st gen.) vs. Xeon v2
64 cores (1 th/core) vs. 61 cores (4 th/core) vs. 16 cores (2 th/core)

**Runtimes of our Stencil Codes (seconds)**

<table>
<thead>
<tr>
<th>Version</th>
<th>Acoustic</th>
<th></th>
<th></th>
<th>Seismic</th>
<th></th>
<th></th>
<th>Heat</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xeon v2</td>
<td>KNC</td>
<td>KNL</td>
<td>Xeon v2</td>
<td>KNC</td>
<td>KNL</td>
<td>Xeon v2</td>
<td>KNC</td>
</tr>
<tr>
<td>Base</td>
<td>392</td>
<td>298</td>
<td>33</td>
<td>647</td>
<td>640</td>
<td>87*</td>
<td>270</td>
<td>189</td>
</tr>
<tr>
<td>Vectorized</td>
<td>357</td>
<td>62</td>
<td>24</td>
<td>625</td>
<td>116</td>
<td>66</td>
<td>250</td>
<td>41</td>
</tr>
</tbody>
</table>

KNL in cache mode and (*) means 2 thread/core

**Results Obtained**

- The vectorization process in Xeon Phi is effective:
  - Speedup against their base versions: 4X for KNC and 1.4X for KNL
  - Speedup against Xeon v2: 7X for KNC and 13X for KNL
- Memory system (in Xeon v2 and KNL) is the limiting factor (unable to provide data at the desired rate)
**CASE STUDY: 3-D STENCIL CODES**

**CODE MODERNIZATION: THREAD PARALLELIZATION (collapse)**

**EXPOSING MORE PARALLELIZATION**

- Loop collapse can help to **expose more parallelism**
- Adding the **collapse (2)** modifier to the OpenMP *pragma* that parallelizes the loop
- This modifier **merges** or **fuses** the two outermost loops of the evaluated kernels in a same loop
- This increases the number of work units that can be given to each thread

---

**3-D STENCIL KERNEL PARALLELIZED USING COLLAPSE**

```plaintext
for time = 0; time < TimeMax; time ++ do
    #pragma omp parallel for collapse (2);
    for z = 1; z < depth - BorderSize; z ++ do
        for y = 1; y < height - BorderSize; y ++ do
            for x = 1; x < width - BorderSize; x ++ do
                stencil_solver_kernel();
            end for
        end for
    end for
    tmp = Input_Grid; Input_Grid = Output_Grid; Output_Grid = tmp;  
end for
```

José M. García
CASE STUDY: 3-D STENCIL CODES
CODE MODERNIZATION: THREAD PARALLELIZATION (scheduling)

Loop Scheduling & load balance

- The scheduling policy: defines how the iterations of the loop are distributed between the threads
- Balanced load: divides the work between the threads in an equitable way
- There are four types of scheduling available at compile time (static, dynamic, guided and auto)
- Added the schedule modifier to OpenMP pragma parallel (e.g., #pragma omp parallel for schedule(type[,size]))

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Threads</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>0</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16 17 18 19 20 21 22 23</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>24 25 26 27 28 29 30 31</td>
</tr>
<tr>
<td>dynamic</td>
<td>0</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3 5 6 7 8 9 10 11</td>
</tr>
<tr>
<td>guided</td>
<td>0</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4 5 6 7 8 9 10 11</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12 13 14 15 22 23</td>
</tr>
</tbody>
</table>

from Colfax HowTo slides
Evaluation of Scheduling policies on KNC

![Graph showing evaluation of scheduling policies on KNC]
**CASE STUDY: 3-D STENCIL CODES**

**CODE MODERNIZATION: EVALUATION OF THREAD PARALLELIZATION**

Evaluation of Scheduling policies on KNC

- **static** is not as good as supposed
  - ⇒ Some work imbalance among the different cores
- **guided** improves by 5.51% when compared to **static**
- The best performing policy was **dynamic**:
  - schedule(dynamic, 4) ⇒ 34% acceleration for the acoustic
  - schedule(dynamic, 4) ⇒ 42% for the seismic
  - schedule(dynamic, 2) ⇒ 30% for the heat
**CASE STUDY: 3-D STENCIL CODES**

**CODE MODERNIZATION: EVALUATION OF THREAD PARALLELIZATION**

Evaluation of Scheduling policies on KNL

- Cache mode with 64 threads

- For KNL, the variability between configurations is minimal

- `dynamic` was the best option for acoustic

José M. García
Affinity Policies

- **Bind logical threads to specific physical cores**
- **3 types of affinity:** compact, scatter and balanced

- Setting affinity prevents thread migration
- Set up with the environment variable `KMP_AFFINITY`
Evaluation of affinity policies on KNC

Affinity has a little influence on the execution time.
**Case study: 3-D stencil codes**

**Code modernization: Evaluation of thread parallelization**

Evaluation of affinity policies on **KNL**

![Graph showing performance with different affinity policies](chart)

Cache mode with 64 threads

For KNL, the **compact** parameter is the worst option for all cases.
Loop Tiling: Cache Blocking

- **Blocking** is a transformation which **groups loop iterations** into subsets of size \( N \) to improve data locality.

- The first step was to create **tiles** of sizes \( \text{width}_{\text{Tblock}} \), \( \text{height}_{\text{Tblock}} \) and \( \text{depth}_{\text{Tblock}} \) (for dimension \( X \), \( Y \) and \( Z \), respectively).

- Then, **three additional loops** were created over the three existing loops to traverse the dataset in the tiles of the selected sizes.

*From Colfax HowTo slides*
Case study: 3-DStencil codes
Code modernization: Memory optimization

Blocking technique applied to the 3-D Stencil solver

1: for $bz = 1; bz < depth - BorderSize; bz += depth_Tblock$ do
2: for $by = 1; by < height - BorderSize; by += height_Tblock$ do
3: for $bx = 1; bx < width - BorderSize; bx += width_Tblock$ do
4: for $z = bz; z < MIN(bz + depth_Tblock, depth - BorderSize); z += do$
5: for $y = by; y < MIN(by + height_Tblock, height - BorderSize); y += do$
6: for $x = bx; x < MIN(bx + width_Tblock, width - BorderSize); x += do$
7: stencil_solver_kernel();
8: end for
9: end for
10: end for
11: end for
12: end for
13: end for
Case study: 3-D Stencil codes

Code modernization: Evaluation of memory optimization

Analyzing YZ blocking size - KNC

Blocking on the Y and Z axes

Best block size: height_Tblock equals to 4, and depth_Tblock also equals to 4
Analyzing X blocking size - KNC

Blocking on the X axis

- Setting the block size to 4 (Y axis) and 4 (Z axis)
- Best block size: width_Tblock equals to 200
**CASE STUDY: 3-D STENCIL CODES**

**CODE MODERNIZATION: EVALUATION OF MEMORY OPTIMIZATION**

Analyzing blocking size - **KNL**

- Firstly, the best blocking configuration (200,4,4) was used varying the number of threads

Other block size configurations were analysed
- Results show negative effects on both scalar and vector codes
**CASE STUDY: 3-D STENCIL CODES**

**CODE MODERNIZATION: SUMMARY**

Xeon Phi KNL (2nd gen.) vs. Xeon Phi KNC (1st gen.) vs. Xeon v2
64 cores (1 th/core) vs. 61 cores (4 th/core) vs. 16 cores (2 th/core)

### PERFORMANCE OF OUR *Stencil* CODES (GFLOPs)

<table>
<thead>
<tr>
<th>Version</th>
<th>Acoustic</th>
<th>Seismic</th>
<th>Heat</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xeon v2</td>
<td>KNC</td>
<td>KNL</td>
</tr>
<tr>
<td>Base</td>
<td>8.7</td>
<td>10.4</td>
<td>81</td>
</tr>
<tr>
<td>+ Vectorized</td>
<td>8.8</td>
<td>51.2</td>
<td>112</td>
</tr>
<tr>
<td>+ Scheduling</td>
<td>-</td>
<td>69.7</td>
<td>138</td>
</tr>
<tr>
<td>+ Affinity</td>
<td>-</td>
<td>72.7</td>
<td>138</td>
</tr>
<tr>
<td>+ Blocking</td>
<td>-</td>
<td>76.2</td>
<td>110</td>
</tr>
</tbody>
</table>

**KNL in cache mode**

- **Speedups** (in execution time):
  - 8-11X between KNC and Xeon v2 (parallel base version)
  - 15-20X between KNL and Xeon v2 (parallel base version)

- KNL obtains performance improvements between 1.5 to 1.7 to KNC
- KNL: Stencil codes do not benefit from blocking
Case study: 3-D Stencil Codes

Conclusions

Lessons learned

- Code portability: Stencil kernels have been programmed in C/C++ with OPENMP extensions. Developers only need to re-compile the codes and they run in our Intel multi- and many-core target platforms immediately (for KNC the compiler option (-mmic)

- The process of code modernization focuses on three areas: vectorization, parallelization and memory traffic reduction (bandwidth tuning)

- Vectorization: Intel icc compiler and look at the vectorization report

- Vectorization: data rearranged, data aligned and padding. Added the directive #pragma ivdep

- Parallelization: Added #pragma omp parallel for with the modifier collapse (2) and schedule (dynamic)

- Parallelization: Affinity balanced (KNC) and scatter (KNL)

- Exploiting blocking in the X, Y, Z axes also leads to additional performance gains for all kernels in KNC, but no in KNL (cache mode)

- Speedups (in execution time):
  - Xeon Phi KNC obtains up to 11X over Xeon v2
  - Xeon Phi KNL (cache mode) obtains up to 20X over Xeon v2
Motivation Background Stencil Web ACO Conclusions & Future

Founda. Vector Parallel Memory

CASE STUDY: 3-D STENCIL CODES
PAPERS PUBLISHED


SPEEDING UP SCIENTIFIC CODES IN HPC ARCH.
TOMORROW TALK CONTENTS

- **Background**
  - Parallelism: Technology trends and parallel programming
  - Intel high-end architectures (multicores & manycores)
  - Code modernization: Best practices

- **Practical examples**
  - Stencil codes: Scientific apps that operate over an N-dimensional data structure that changes over time, given a fixed computational pattern
  - **Semantic Web**: A Semantic dataset generator that transforms relational or XML data into semantic repositories
  - **Ant Colony Optimization (ACO)**: A Bio-inspired metaheuristic applied to a wide range of NP-hard combinatorial optimization problems

- **Conclusions and Lessons learned**
- **Future lines**: Domain-Specific Languages (DSLs) and Domain-Specific Architectures (DSAs)
Speeding up Scientific Codes in HPC Architectures by Code Modernization: Lessons Learned (1/2)

José M. García
jmgarcia@um.es

Parallel Computing Architecture Group (GACOP)
University of Murcia
Murcia (Spain)

Academic Training Lecture Programme @ CERN
Geneve (Switzerland), June 2019
**Motivation**

Background

Stencil

Parallel Memory

Web

ACO

Conclusions & Future

José M. García

jmgarcia@um.es

Parallel Computing Architecture Group (GACOP)

University of Murcia

Murcia (Spain)

Academic Training Lecture Programme @ CERN

Geneve (Switzerland), June 2019
SPEEDING UP SCIENTIFIC CODES IN HPC ARCH.

YESTERDAY TALK CONTENTS

- **Background**
  - Parallelism: Technology trends and parallel programming
  - Intel high-end architectures (multicores & manycores)
  - Code modernization: Best practices

- **Practical examples**
  - Stencil codes: Scientific apps that operate over an N-dimensional data structure that changes over time, given a fixed computational pattern
  - Semantic Web: A Semantic dataset generator that transforms relational or XML data into semantic repositories
  - Ant Colony Optimization (ACO): A Bio-inspired metaheuristic applied to a wide range of NP-hard combinatorial optimization problems

- **Conclusions and Lessons learned**

- **Future lines**: Domain-Specific Languages (DSLs) and Domain-Specific Architectures (DSAs)
**BACKGROUND**

**SUMMARY**

**TECHNOLOGY TRENDS & PARALLEL PROGRAMMING**

- Processor architecture is composed of multiple cores
  - Exploits **data parallelism** (SIMD) using vector instructions
  - Exploits **thread parallelism** (TLP) among cores
- Parallel issues (i.e., **memory locality**, granularity, coordination and synchronization, etc) make parallel programming even harder than sequential programming

**INTEL HIGH-PERFORMANCE ARCHITECTURES**

- **Intel Xeon multicore**: High single-thread performance, **AVX-2** instruct. (256 bits vector unit wide), few cores/socket (8 to 20)
- **Intel Xeon Phi**: Low single-thread performance, **AVX-512** instruct. (512-bits vector unit wide), many cores/socket (61 to 68)
- Intel profiling tools

**CODE MODERNIZATION: BEST PRACTICES**

- Take full advantage of **modern hardware**: manual approach
- **Single node**: Scalar, vectorization, parallelization and memory tuning
OUTLINE

1 BACKGROUND

2 CASE STUDY: 3-D STENCIL CODES

3 CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

4 CASE STUDY: ACO

5 CONCLUSIONS AND FUTURE RESEARCH LINES
THE PROBLEM

- SWIT (Semantic Web Integration Tool) was a tool developed by the TECNOMOD research group (from the University of Murcia)
- SWIT transforms relational or XML data into repositories in Semantic Web formats (RDF or OWL)
- This tool was developed in the frame of The Quest for Orthologs
- Written in Java, it required more than a month of computational hours to transform certain databases

The goal was to improve the execution time of SWIT

CODE MODERNIZATION FEATURES

- A case of moving from interpreted to compiled language
- A task-based parallelization example
- Also I/O bounded application
- Target platform: Xeon v4 (40 cores, 2 threads/core)
The Quest for Orthologs consortium

- The Quest for Orthologs (QfO) is a joint effort to improve and standardize orthology predictions through collaboration and the use of shared reference datasets.
- ‘Orthology’ is the identification of gene relationships: Any of two or more homologous gene sequences found in different species related by linear descent.
- More than 40 different databases in XML format.
- Semantic web techniques are used to data normalization and integration:
  - They offer a natural space for data integration and interoperability.
  - Ontologies are the cornerstone technology: the OWL language.
  - Linked Open Data is a Semantic Web initiative for publishing and sharing the web content in a semantic format like RDF.
SWIT (Semantic Web Integration Tool)

- SWIT provides semantics-rich, ontology-driven transformation and integration of datasets (http://sele.inf.um.es/swit/)
- The major performance limitation is the application of identity rules in data integration scenarios (for large datasets)
An input example (XML/relational database)

```xml
<species name="Escherichia−coli" NCBItaxId="83333">
  <genes>
    <gene id="1" protId="P07118" genelD="valS"/>
  </genes>
</species>

<species name="Nematocida−parisii" NCBItaxId="881290">
  <genes>
    <gene id="2" protId="I3EQN8" genelD="NEPG_00863"/>
  </genes>
</species>

<groups>
  <orthologGroup id="1">
    <geneRef id="1"/>
    <geneRef id="2"/>
  </orthologGroup>
</groups>
```

An output example (RDF/OWL dataset)

```xml
<rdf:Description rdf:about="http://identifiers.org/gene/83333/valS">
  <dct:identifier rdf:datatype="http://www.w3.org/2001/XMLSchema#string">valS</dct:identifier>
  <obo:taxonomy rdf:resource="http://identifiers.org/taxonomy/83333"/>
  <sio:synthesize rdf:resource="http://purl.org/net/orth#protein/sIO_000750_0/P07118"/>
  <rdf:type rdf:resource="http://purl.org/net/orth#Gene"/>
</rdf:Description>
```
3 orthology databases have been used from https://questfororthologs.org/orthology_databases

**Inparanoid**: This resource stores orthology relations between genes from different species. We have used the InParanoid files for the species *S. pombe*, *C. elegans* and *G. gorilla*, whose sizes are 49 MB, 318 MB and 371 MB. These three data collections include 50, 233 and 174 files respectively.

**TreeFam**: This resource stores groups of orthologs for several genomes. We have used the whole database, which is distributed in one 612 MB file.

**OMA**: This resource also stores groups of orthologs for several genomes. We have used the whole database, which is distributed in one 1.5 GB file.

---

[a](http://inparanoid.sbc.su.se/download/8.0_current/Orthologs_OrthoXML/)
[b](http://www.treefam.org/download)
[c](https://omabrowser.org/oma/current/)
## Output Data Generated (RDF Triples)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Input and output instances</th>
<th>Triples generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>InParanoid - S.pombe</td>
<td>326,379</td>
<td>829,152</td>
</tr>
<tr>
<td>InParanoid - C.elegans</td>
<td>2,155,382</td>
<td>5,385,137</td>
</tr>
<tr>
<td>InParanoid - G.gorilla</td>
<td>2,511,846</td>
<td>6,144,129</td>
</tr>
<tr>
<td>InParanoid - Whole DB</td>
<td>295,885,160</td>
<td>440,025,733</td>
</tr>
<tr>
<td>OMA</td>
<td>16,641,865</td>
<td>52,068,297</td>
</tr>
<tr>
<td>TreeFam</td>
<td>2,720,491</td>
<td>14,803,371</td>
</tr>
</tbody>
</table>

The transformation of the 43 GB of the InParanoid database with identity rules required of 919 computational hours (around 38 days)!!!
The HPC-SWIT version has been fully reimplemented from scratch in C/C++

Additionally, the #pragma omp simd was added in some loops to avoid an incorrect data dependence or a multi-versioned chunk of code

The original version used SPARQL queries to detect redundant data, which is not efficient for large datasets or for identity rules with many conditions

HPC-SWIT uses two new data structures: hash maps of vectors

A hash map for AND conditions and another one for OR conditions

The new method generates nearly unique hashes per each individual (depending on the type of rule)
CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS
CODE MODERNIZATION: REDESIGNING SWIT

Xeon v4
Using a single core (and one thread)

SWIT execution comparison using 3 datasets of InParanoid database

Original V. W.I and HPC V. W.I stands for the original version and optimized version of SWIT with identity rules, whilst Original V. and HPC V. N.I denotes the usage of these versions when no identity rules are applied.
Xeon v4
Using a single core (and one thread)

A speedup between 3 and 4.5 is obtained without identity rules
A speedup between 209 and 671 is obtained while using identity rules
Xeon v4
Using a single core (and one thread)
## Single Core Execution with Identity Rules: Speed-up Table

<table>
<thead>
<tr>
<th>Dataset</th>
<th>HPC-SWIT vs SWIT Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>InParanoid <em>S.pombe</em></td>
<td>209x</td>
</tr>
<tr>
<td>InParanoid <em>C.elegans</em></td>
<td>396x</td>
</tr>
<tr>
<td>InParanoid <em>G.gorilla</em></td>
<td>671x</td>
</tr>
<tr>
<td>InParanoid <em>WB</em></td>
<td>240x</td>
</tr>
<tr>
<td>OMA</td>
<td>1,395x</td>
</tr>
<tr>
<td>TreeFam</td>
<td>12,606x</td>
</tr>
</tbody>
</table>
Some databases (e.g. InParanoid) have many input files

Our parallelization strategy consisted in setting one input file and one HPC-SWIT instance per core ⇒ task parallelization

The parallelization is applied at process level by using the GNU *Parallel* tool

The following script shell is run:

```
[user@ibsen ~]$ files = ( $( find $dir -maxdepth 1 -type f) )
[user@ibsen ~] parallel ./ swit {} " arguments " ::: ${ files[@]}
```
An additional improvement factor of 4 was achieved

The original SWIT algorithm required 38 days to process the whole InParanoid database (43 GB). Parallel HPC-SWIT in less than 1 hour ($\approx 55$ minutes).
### Parallel HPC-SWIT vs SWIT Speed Up

<table>
<thead>
<tr>
<th>Database</th>
<th>Dataset</th>
<th>Sequential speed up</th>
<th>Parallel speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>InParanoid</td>
<td><em>S. pombe</em></td>
<td>209x</td>
<td>1,964x</td>
</tr>
<tr>
<td></td>
<td><em>C. elegans</em></td>
<td>397x</td>
<td>6,196x</td>
</tr>
<tr>
<td></td>
<td><em>G. gorilla</em></td>
<td>672x</td>
<td>11,187x</td>
</tr>
<tr>
<td>Whole database</td>
<td></td>
<td>240x</td>
<td>1,003x</td>
</tr>
<tr>
<td>OMA</td>
<td></td>
<td>1,395x</td>
<td>318x</td>
</tr>
<tr>
<td>TreeFam</td>
<td></td>
<td>12,606x</td>
<td>6,581x</td>
</tr>
</tbody>
</table>

### Remarks

We split OMA and TreeFam into several files. However, their parallel speed up is lower due to problems with the structure of OrthoXML format. Each OrthoXML file contains one node per species, which contains its respective list of genes. Splitting an OrthoXML file requires to replicate this information increasing the data size to process.
**HARD DRIVE DISK BOTTLENECK**

- This is due to **write large files** in a short period of time

- **Solution**: **Compress** the data before sending it to disk. The compression method used is the “Deflate” function applied in ZIP files (Huffman coding y LZ77)
**Compression method**

- DEFLATE algorithm, standard zip
- A compression ratio of $\approx 27x$ for RDF/OWL files

**HPC-SWIT: I/O Execution time for InParanoid database**

<table>
<thead>
<tr>
<th>Version</th>
<th>Compression</th>
<th>Time (hrs)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>No</td>
<td>919 ($\approx 38$ days)</td>
<td>Not applicable</td>
</tr>
<tr>
<td>HPC-SWIT</td>
<td>No</td>
<td>0.91 ($\approx 55$ m)</td>
<td>1,003x</td>
</tr>
<tr>
<td>HPC-SWIT</td>
<td>Yes</td>
<td>0.14 ($\approx 8$ m 56 s)</td>
<td>6,173x</td>
</tr>
</tbody>
</table>
Xeon v4
40 cores (2 th/core)

**HPC-SWIT** execution times varying the thread count, w/ or w/o compression

- **No Compression**
- **Enabling Compression**

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>60</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>13819.67</td>
<td>6045.67</td>
<td>4323.67</td>
<td>3746</td>
<td>3648.33</td>
<td>3463.33</td>
<td>3515.33</td>
<td>3319</td>
<td>3318</td>
<td>3300.33</td>
</tr>
<tr>
<td></td>
<td>14255</td>
<td>5122.33</td>
<td>2892.67</td>
<td>1703.67</td>
<td>1435.33</td>
<td>853.67</td>
<td>601.33</td>
<td>572</td>
<td>543.33</td>
<td>536.33</td>
</tr>
</tbody>
</table>
HPC-SWIT execution times varying the thread count, w/ or w/o compression

Compression trade offs

- **Compression** might be adding up an excessive **overhead** with reduced number of threads
- Its effectiveness depends on the size of the input/output dataset, number of files, and disk technology (HDD, SSD, etc.)
LESSONS LEARNED

- HPC-SWIT (https://bitbucket.org/Neobernad/swit-test) has been written in C++ enabling vector capabilities with pragmas. The checking of identity rules was implemented using hash maps of vectors both for AND and OR conditions.

- In a single core, HPC-SWIT run faster than SWIT for both with and without checking identity rules.

- A parallel implementation was developed for databases with many input files.

- We followed the task parallelization strategy that consisted in setting one input file and one HPC-SWIT instance per core, doubling the performance benefits.

- We realized that accesses to HDD were the bottleneck. We implemented output data compression obtained additional performance benefits depending on the dataset size and the technology used for storage.

- Speedups with identity rules on the Xeon v4 (in execution time):
  - InParanoid database: 240X (single core), 1,000X (parallel, 80 th), 6,200 (parallel, 80 th & I/O compression ratio of 27X)
  - OMA database: 1,395X (single core), worse in parallel
  - TreeFam database: 12,606X (single core), worse in parallel

OUTLINE

1 BACKGROUND

2 CASE STUDY: 3-D STENCIL CODES

3 CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

4 CASE STUDY: ACO

5 CONCLUSIONS AND FUTURE RESEARCH LINES
Case Study: ACO Presentation

The Problem

- The Ant Colony Optimization (ACO) is a bio-inspired metaheuristic applied successfully to a wide range of NP-hard combinatorial optimization problems.
- Many real-world problems can be reduced to them, e.g., route scheduling, goods dispatching, etc.
- First proposed by Marco Dorigo in 1992 and based on ants’ foraging process.
- Requires a lot of computations (compute-bound problem).

- The goal is improving the ACO’s execution time and testing its scalability (parallel efficiency) in high-end Intel architectures.

Code Modernization Features

- The base code is based on the sequential Stützle’s implementation in C++.
- Best practices: scalar, parallelization, vectorization, and memory.
- Evaluation of parallel efficiency (or scalability).
- Target platforms: Xeon v2 (16 cores, 2 threads/core), Xeon v4 (40 cores, 2 threads/core), KNC (61 cores, 4 threads/core) and KNL (68 cores, 4 threads/core).
**CASE STUDY: ACO**

**BACKGROUND**

**GENERAL STRUCTURE OF ACO ALGORITHMS**

1. *Initialization()*
2. **while** not *TerminationCondition()** **do**
3.   *TourConstruction()*
4.   *PheromoneUpdate()*
5. **end while**
**CASE STUDY: ACO**

**BACKGROUND**

**ACO applied to the Travelling Salesman Problem (TSP)**

- Consists of finding the *shortest round trip* tour that include at least once each city from a set of \( n \) cities.
- The TSP is a paradigmatic NP-hard combinatorial optimization problem.
- The symmetric TSP has been used, in which the distance between two cities, \( i \) and \( j \), is the same in both directions (\( d_{ij} = d_{ji} \)).
- The tour construction stage takes over 99.8% of the time.

**The ACO Tour Construction Stage**

```plaintext
1: for a = 1 to m do
2:   {Place ant on initial city}
3:   initial_city ← choose_initial_city()
4:   tour[a][1] ← initial_city
5:   visited[a][initial_city] ← true
6: {Construct tour}
7: for step = 2 to n do
8:   choose_next(a, step)
9: end for
10: tour[a][n] ← tour[a][1]
11: tour_length[a] ← compute_tour_length(tour[a])
12: end for
```
ANT SYSTEM VARIANT

- In Ant System, at the start of the tour construction stage, each ant is placed on a randomly chosen initial city.
- At each construction step, each ant makes use of a probabilistic action choice rule, called random proportional rule, in order to choose its next city to visit.
- \( \tau_{ij} \) is the amount of pheromone associated with edge \((i, j)\), \( \eta_{ij} = 1/d_{ij} \) is a distance value computed a priori, \( \alpha \) and \( \beta \) are two parameters (fixed at the beginning of an execution), and \( N_i^k \) is the non-tabu list.

\[
P_{k}^{ij} = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}
\]
**CASE STUDY: ACO**

**BACKGROUND**

### Roulette Wheel Selection

**Input:** Ant identifier \((a)\), construction step \((phase)\).

1. \(current\_city \leftarrow tour[a][phase - 1]\)
2. \{Selection Probabilities Computation\}
3. \(prob\_sum \leftarrow 0\)
4. \(for\ i = 1\ to\ n\ do\)
5. \(\quad\ if\ visited[a][i]\ then\)
6. \(\quad\quad\ prob[i] \leftarrow 0\)
7. \(\quad\ else\)
8. \(\quad\quad\ prob[i] \leftarrow choice\_info[current\_city][i]\)
9. \(\quad\quad\ prob\_sum \leftarrow prob\_sum + prob[i]\)
10. \(\quad\ end\ if\)
11. \(\end for\)
12. \{City Selection\}
13. \(r \leftarrow random(0..prob\_sum)\)
14. \(city \leftarrow 1\)
15. \(partial\_sum \leftarrow prob[city]\)
16. \(while\ partial\_sum < r\ do\)
17. \(\quad\ city \leftarrow city + 1\)
18. \(\quad\ partial\_sum \leftarrow partial\_sum + prob[city]\)
19. \(\end while\)
20. \(tour[a][phase] \leftarrow city\)
21. \(visited[a][city] \leftarrow true\)

**Roulette Wheel Selection (default)**

- Each not visited city is assigned to a portion (proportionally to its probability) on a circular **roulette wheel**.
- A **random number** is generated, and the portion in which the number takes place determines the selected city.

---

José M. García  
Code Modernization  
CERN - June 2019  
91/124
Experimental methodology

- Our different implementations are tested using a set of instances from the **TSPLIB benchmark library**
- ACO parameter settings: $m = n$ (where $m$ is the number of ants and $n$ is the number of cities), $\alpha = 1$ and $\beta = 5$.
- Performance figures are given for **single-precision numbers**, and the execution times shown are the average of 10 independent runs.
- Xeon Phi KNC and KNL have been set to **balanced** affinity, and Xeon v2 and v4 to **compact** affinity.
- On Xeon Phi KNL, the experiments are performed on **flat mode** (both DDR4 and MCDRAM).
CASE STUDY: ACO
CODE MODERNIZATION: SCALAR OPTIMIZATIONS

SCALAR OPTIMIZATIONS

- Avoid repetitive computations using previously calculated results. \([\eta_{ij}]^\beta\) is pre-computed at the beginning of the program and stored in a matrix.
- Use the right precision for built-in functions (e.g., replace `pow()` with `powf()`).
- Avoid runtime auto-promotion and type conversions.
- Replace costly arithmetic expressions with others of lower cost (e.g., replace divisions with multiplications by the inverse).

Especially useful in our low single-thread performance architectures (e.g. KNC).
PARALLELIZATION STRATEGY

- The tour construction stage is inherently parallel, as each ant can construct its solution individually.
- Map ants to threads (parallelizing the outer loop with OpenMP).
- Handle data structures.

TOUR CONSTRUCTION

1: #pragma omp parallel for
2: for a = 1 to m do
3:    choose_initial_city(a)
4: for step = 2 to n do
5:    choose_next_city(a, step, thread_id) → Selection function (99% of the time)
6: end for
7: compute_tour_length(a)
8: end for
CASE STUDY: ACO
CODE MODERNIZATION: THREAD PARALLELIZATION

Xeon Phi KNC (1st gen.) vs. Xeon v2
61 cores (4 th/core) vs. 16 cores (2 th/core)

Parallel efficiency (or scalability) for tour construction
Applying Best Practices

- **Data alignment**: We have used `_mm_malloc(size, 64)` instead of `malloc()` for data alignment.

- **Align padding**: We have padded the inner dimension of multi-dimensional arrays to guarantee alignment for each row of the matrix.

- **Data alignment hints**: Concretely, we have used `__assume_aligned(ptr, 64)` for pointers. This clues are provided in the region of the code where data structures are used within a loop.
CASE STUDY: ACO
CODE MODERNIZATION: VECTORIZATION

Vectorization report for the main loops of Roulette Wheel Selection

PROBLEMS APPEARED

Looking at the vectorization report from the Intel compiler, we noticed that none of the two loops in the Roulette Wheel Selection were vectorized.

Report from: Loop nest, Vector & Auto—parallelization optimizations [loop, vec, par]

LOOP BEGIN at ants.inc(237,5)
remark #15344: loop was not vectorized: vector dependence prevents vectorization
remark #15346: vector dependence: assumed FLOW dependence between prob (239:13) and choice_info (241:13)
remark #15346: vector dependence: assumed ANTI dependence between choice_info (241:13) and prob (239:13)
remark #25439: unrolled with remainder by 2
LOOP END

LOOP BEGIN at ants.inc(252,9)
remark #15523: loop was not vectorized: loop control variable city was found, but loop iteration count cannot be computed before executing the loop
LOOP END
**CASE STUDY: ACO**

**CODE MODERNIZATION: VECTORIZATION**

---

**ROULETTE WHEEL SELECTION**

**Input:** Ant identifier \((a)\), construction step \((phase)\).

1. \(current\_city \leftarrow tour[a][phase - 1]\)
2. \{Selection Probabilities Computation\}
3. \(prob\_sum \leftarrow 0\)

4. \(\text{for } i = 1 \text{ to } n \text{ do}\)
   5. \(\text{if } visited[a][i] \text{ then}\)
      6. \(prob[i] \leftarrow 0\) \(\rightarrow\) Not vectorized, but solvable
      7. \(\text{else}\)
      8. \(prob[i] \leftarrow choice\_info[current\_city][i]\)
      9. \(prob\_sum \leftarrow prob\_sum + prob[i]\)
   10. \(\text{end if}\)
11. \(\text{end for}\)

12. \{City Selection\}
13. \(r \leftarrow \text{random}(0..prob\_sum)\)
14. \(city \leftarrow 1\)
15. \(partial\_sum \leftarrow prob[city]\)

16. \(\text{while } partial\_sum < r \text{ do}\)
   17. \(city \leftarrow city + 1\) \(\rightarrow\) Inherently sequential
   18. \(partial\_sum \leftarrow partial\_sum + prob[city]\)
19. \(\text{end while}\)

20. \(tour[a][phase] \leftarrow city\)
21. \(visited[a][city] \leftarrow \text{true}\)

---

José M. García

Code Modernization

CERN - June 2019
Case study: ACO

Code modernization: Vectorization

**Problems appeared**

- First loop: computes the probability of selection for each city ⇒ Vector dependence and if statement

- Second loop: simulates the roulette spinning ⇒ the number of iterations is not known at compilation time, and each iteration depends on the previous one
**ALTERNATIVE SELECTION FUNCTION: V-ROULETTE**

- Use `#pragma ivdep` to ignore vector dependences
- Add a tabu list and replace the *if sentence* for a multiplication
  ⇒ first loop vectorized

---

**1) Selection Probabilities Computation**
(vectorized)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.15</td>
<td>1</td>
</tr>
</tbody>
</table>

**2) City Selection**
(serial)

- `r ← random(0..prob_sum)`
- `city ← 1`
- `partial_sum ← prob[city]`
- **while** `partial_sum < r` **do**
  - `city ← city + 1`
  - `partial_sum ← partial_sum + prob[city]`
- **end while**
**ALTERNATIVE SELECTION FUNCTION: I-ROULETTE (INDEPENDENT ROULETTE) v1**

- Use a different random number for each city (independent)
- Change data structures: the seed for generating random numbers needs to be replicated to a matrix of seeds
- The city with the highest weight is selected as the next one
  ⇒ second loop partly vectorized

**Diagram: Alternative Selection Function**

1) Weights Computation (vectorized)
2) City Selection (serial)

<table>
<thead>
<tr>
<th></th>
<th>0.7</th>
<th>1</th>
<th>0.4</th>
<th>0.28</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.1</td>
<td>0</td>
<td>0.6</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.15</td>
<td>1</td>
<td>0.2</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>0.25</td>
<td>0</td>
<td>0.5</td>
<td>0</td>
</tr>
</tbody>
</table>

**Notes:**
- City selection is marked with an X.
- Reduction is calculated by multiplying the chosen weight with the reduction factor.
- City selection is based on the maximum weight.
CASE STUDY: ACO
Code modernization: Vectorization

ALTERNATIVE I-ROULETTE v2: VECTORIZED REDUCTION

- Use `pragma #pragma ivdep` to ignore vector dependences and avoid loop multiversioning
- Automatic reduction vectorization from Intel `icc` compiler version 16

SELECTION FUNCTION: I-ROULETTE v2

Input: Ant identifier (a), construction step (step), thread identifier (thread_id).
Output: Selected city.

```plaintext
1: current_city = tour[a][step - 1]
2: city ← -1
3: max_weight ← -1

#pragma ivdep

4: for i = 1 to n do
5:   w ← choice_info[current_city][i] * visited[a][i] * rand01(seeds[thread_id][i])
6:   if w > max_weight then
7:     city ← i
8:     max_weight ← w
9: end if
10: end for
11: return city
```

VECTORIZATION REPORT
(Intel compiler)

Loop (lines 4-10):
- Vectorized
- Unit stride
- Vector length = architecture’s vector length
Cities are grouped into blocks and each block’s probability is computed as the addition of the probabilities of the cities within that block.

Two roulette wheel selections take place: one for choosing a block, and a second for choosing a city within that block.

1) Selection Probabilities Computation (vectorization within each block)

2) Block Selection (serial)

3) City Selection (serial)
CASE STUDY: ACO

CODE MODERNIZATION: VECTORIZATION

Xeon Phi KNC (1st gen.)
61 cores (4 th/core)

Selection Functions on Xeon Phi KNC

Speed up for the tour construction stage with different selection functions (compared to Roulette Wheel)
**CASE STUDY: ACO**

**CODE MODERNIZATION: WALL-CLOCK TIME EVALUATION**

Xeon Phi KNC (1st gen.) vs. Xeon v2
61 cores (4 th/core) vs. single core (1 th/core)

**SPEED UP FOR THE TOUR CONSTRUCTION STAGE**

Execution time on Xeon Phi KNC compared to **sequential code** on Xeon v2
Case study: ACO
Code modernization: Wall-clock time evaluation

Xeon Phi KNL (2nd gen.) vs. Xeon Phi KNC (1st gen.) vs. Xeon v4 vs. Xeon v2
64 cores (3 th/core) / 61 cores (4 th/core) / 40 cores (2 th/core) / 16 cores (2 th/core)

Execution time (s) for tour construction on different architectures
CASE STUDY: ACO
CODE MODERNIZATION: WALL-CLOCK TIME EVALUATION

Xeon Phi KNL (2nd gen.) vs. Xeon Phi KNC (1st gen.) vs. Xeon v4 vs. Xeon v2
64 cores (3 th/core) / 61 cores (4 th/core) / 40 cores (2 th/core) / 16 cores (2 th/core)

- Intel Xeon v4 outperforms the other architectures when it runs instances of up to 3795 cities (speedup of 10X over Xeon v2)
- KNC and KNL outperform Xeon v4 for larger instances
- Speedups for KNC up to 6X and for KNL(MCDRAM) up to 9X
- For the largest instance Xeon v4 is slightly better than the two Xeon Phi
### CASE STUDY: ACO

**Code modernization: Thread Parallelization**

Parallel Efficiency on Xeon multicore

<table>
<thead>
<tr>
<th>Xeon v2 (16 cores, 2 threads/core)</th>
<th>Xeon v4 (40 cores, 2 threads/core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>Speedup</td>
</tr>
<tr>
<td>1 16</td>
<td>1 16</td>
</tr>
<tr>
<td>2 14</td>
<td>2 14</td>
</tr>
<tr>
<td>4 12</td>
<td>4 12</td>
</tr>
<tr>
<td>6 10</td>
<td>6 10</td>
</tr>
<tr>
<td>8 8</td>
<td>8 8</td>
</tr>
<tr>
<td>10 6</td>
<td>10 6</td>
</tr>
</tbody>
</table>

- Xeon v2 obtains good parallel efficiency for small and medium-sized problem instances (around 80%), but it decreases for larger problem sizes (around 40%)
- Xeon v4 shows worse scalability, ranging from 62% to only 20% for large problem sizes
CASE STUDY: ACO

CODE MODERNIZATION: THREAD PARALLELIZATION

Parallel Efficiency on Xeon Phi KNL

- Xeon Phi KNL with MCDRAM memory achieves a parallel efficiency ratio from 31% to 20%
- Xeon Phi KNL with DDR4 memory achieves a parallel efficiency ratio from 30% to 8%
Xeon Phi KNC achieves the best parallel efficiency, ranging from near 100% for small problems to 70% for larger problem sizes, although it drops to 33% for the largest size.
Case Study: ACO
Code Modernization: Thread Parallelization

Parallel Efficiency: Feasible Explanations

1. Core load unbalance: Limited impact (depending on number of cores and problem size)
2. Memory bandwidth limitations: The key factor
3. NUMA effects on data placement: Not for KNC
Motivation Background Stencil Web ACO Conclusions & Future Foundations Parallel Vector Efficiency

Case Study: ACO
Code Modernization: Thread Parallelization

Memory bandwidth analysis with Vtune

- ACO is asking for the highest memory bandwidth in all the execution time
- For large size problems, ACO changes its behaviour: from compute bounded to memory bounded

José M. García  Code Modernization  CERN - June 2019  110/124
Recent proposal: Using a K-nearest neighbor list

- K-nearest neighbor criteria: neighbors ordered by choice_info values
- We tested several values for K
- Always multiples of 8 ⇒ Final value: groups of 32 neighbours
**CASE STUDY: ACO**

**On-going work: K-nearest neighbor list**

Xeon Phi KNL (2nd gen.) vs. Xeon Phi KNC (1st gen.) vs. Xeon v4 vs. Xeon v2
64 cores (3 th/core) / 61 cores (4 th/core) / 40 cores (2 th/core) / 16 cores (2 th/core)

![Execution time (s) for tour construction on different architectures](chart)

- A huge reduction time: KNC around 10X and the rest over 100X
- As memory bandwidth problems were solved:
  - Xeon v4 was again the best architecture
  - KNL's execution time is the same with DDR4 and MCDRAM
Quality of Solution

Independent of the target platform

Quality of solution measured as tour length for 1000 iterations
LESSONS LEARNED

- The Ant Colony Optimization (ACO) metaheuristic code applied to the Travelling Salesman Problem has been modernized.
- Code modernization best practices applied: scalar, parallelization, vectorization, and memory.
- Vectorization: Intel icc compiler and look at the vectorization report. Problems when vectorizing Roulette Wheel ⇒ Solved by other selection algorithm (I-Roulette v2).
- Parallelization: Added #pragma omp parallel for.
- Poor parallel efficiency for large problem sizes ⇒ Three main problems identified: Core load unbalance, Memory bandwidth limitations and NUMA effects on data placement.
- Analysis with VTune: ACO changes to a memory-bound algorithm.
- Speedups (in execution time):
  - Against sequential base version running on Xeon v2: Xeon v4 up to 90X, KNC up to 80X, and KNL(MCDRAM) up to 100X.
  - Against parallel version running on Xeon v2: Xeon v4 up to 10X, KNC up to 6X, and KNL(MCDRAM) up to 9X.


OUTLINE

1 Background

2 Case Study: 3-D Stencil Codes

3 Case Study: Semantic Web and Bioinformatics

4 Case Study: ACO

5 Conclusions and Future Research Lines
Conclusions & Future Work
Lessons Learned

**Code Modernization**

- **Compilers** often **cannot** do the job
  - Automatic parallelization/vectorization still unsolved
  - Often intricate changes in the algorithm required
  - Fast code can be large and *could* violate “good” software engineering practices
- **Portability**: Intel high-end architectures offer code portability with performance gains
- **Code modernization best practices** (single node)
  - Vector instructions
  - Thread parallelization
  - Memory hierarchy
  - Manual tuning required
- **Good speedups** obtained (sometimes very good), but parallel efficiency is more difficult
- **Code modernization** requires expert knowledge in algorithms, coding, and architecture
CONCLUSIONS & FUTURE WORK

Lessons Learned

FROM OUR CASE STUDIES

3-D stencil codes
- Vectorization is the key strategy
- Expose more parallel opportunities using the modifier `collapse (2)` and `schedule (dynamic)`
- The application of blocking techniques improves memory locality for these kernels

HPC-SWIT tool
- Great benefit from an interpreted to a compiled language
- A task-based parallelization strategy
- I/O bottleneck solved by data compression

ACO applied to TSP
- Changes in the code needed for vectorization
- The compute-bound problem changed to a memory-bound for large instance sizes
- Parallel efficiency was affected by core load unbalance, memory bandwidth and NUMA effects
**CONCLUSIONS & FUTURE WORK**

**Dennard scaling + Amdahl’s law**

![Graph](image)

**Figure:** Speedup versus % "Serial" Processing Time; from Hennessy’s talk [2018]
## Conclusions & Future Work

### DSLs & DSAs

### Challenges Ahead

- **Application focus shifts**: From desktop to individual, mobile devices and ultrascale cloud computing, IoT, Big Data, Deep Learning: new constraints
- **Demand for higher performance**: focused on such specific domains
- **HW-approach**: Only path left is Domain Specific Architectures. Just do a few tasks, but extremely well
- **Domain Specific Architectures (DSAs)**: Achieve higher efficiency by tailoring the architecture to characteristics of the domain
- The biggest concern for Exascale application developers is the need to write and maintain multiple versions of their software and the uncertainty over what the architectures will be

### Solutions

- **Domain Specific Languages (DSL)** have to be architecture-independent (so, interesting compiler challenges will exist)
- **Combination**: DSLs architecture agnostic & DSAs

**Real** modern code: One Code (Optimized, portable and future-proof) for All Platforms
CREDITS

PEOPLE (GACOP GROUP)

- José María (Chema) Cecilia (former PhD student, now @ UCAM (Spain))
- Mario Hernández (former PhD student, now @ UA de Guerrero (México))
- Victor Montesinos (former Master student, now @ EPFL (Switzerland))
- José Antonio Bernabé (PhD student)
- Eduardo José Gómez (Master student)
- Pablo Martinez (Master student)
- ... and all the rest of the GACOP group
Spanish MCIU and AEI, as well as European Commission FEDER funds, under grants RTI2018-098156-B-C53 and TIN2016-78799-P (AEI/FEDER, UE)

HiPEAC Collaboration funds & HiPEAC Collaboration funds under grants IST-287759 and IST-687698

Fundación Séneca (Regional Research Agency of Murcia) under grant 18946/JLI/13

Mario Hernández was supported by the PROMEP under the Teacher Improvement Program (UAGro-197) México

“More Data, More Science and ... Moore’s Law?” (and other talks). Prof. Kathy Yelick. EECS and LBNL, UC Berkeley

John Hennessy and David Patterson 2017 ACM A.M. Turing Award Lecture. Los Angeles (USA), June 2018

Talks and lectures from Prof. James Demmel. Computer Science Division. Department of Mathematics. UC Berkeley

Any questions?

**Speeding up Scientific Codes in HPC Architectures by Code Modernization: Lessons Learned (2/2)**

José M. García  
jmgarcia@um.es  

Parallel Computing Architecture Group (GACOP)  
University of Murcia  
Murcia (Spain)

Academic Training Lecture Programme @ CERN  
Geneve (Switzerland), June 2019