SPEEDING UP SCIENTIFIC CODES IN HPC ARCHITECTURES BY CODE MODERNIZATION: LESSONS LEARNED (1/2)

José M. García

jmgarcia@um.es

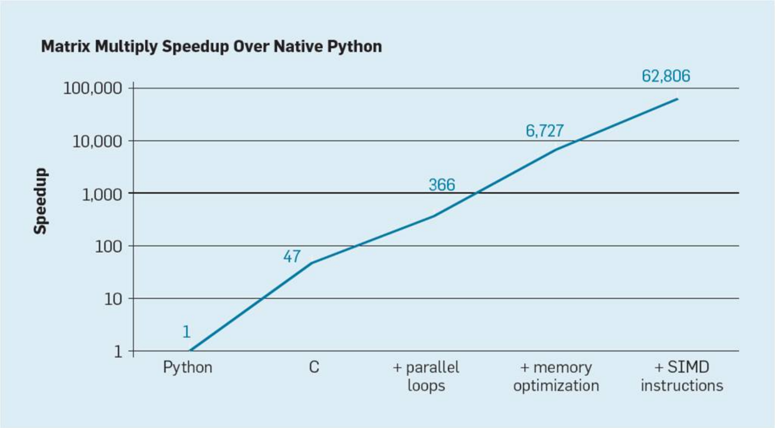
Parallel Computing Architecture Group (GACOP)
University of Murcia
Murcia (Spain)

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

MOTIVATION

A FIRST IMPORTANT FACT

The importance of speeding-up programs[1] :



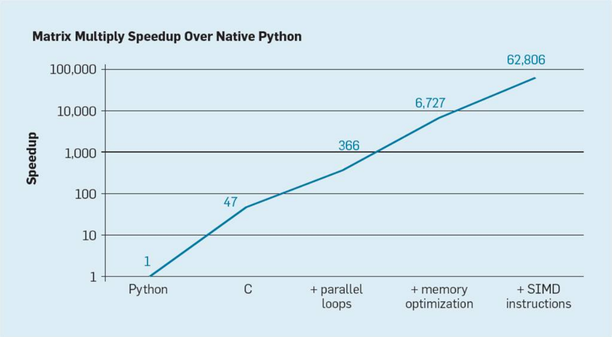
"There's Plenty of Room at the Top," *Leiserson, et. al.*, to appear

---

[1] "*A new Golden Age for Computer Architecture*", J. Hennessy and D. Patterson in the 2018 ACM A.M. Turing Award Lecture

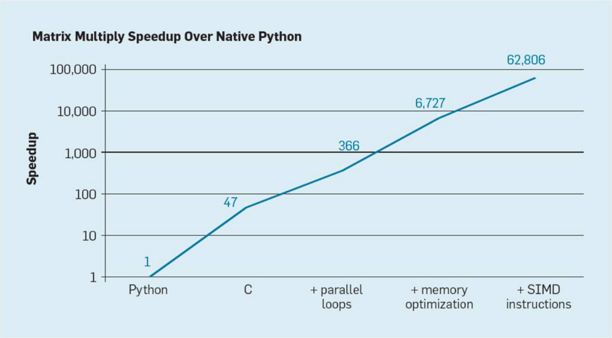
MOTIVATION

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

MOTIVATION
A FIRST IMPORTANT FACT



- 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

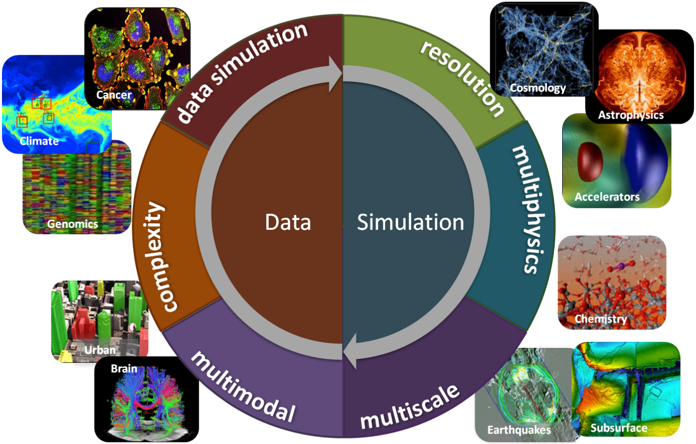
- **Goal**: Build a High-Performance Computer (Supercomputer) that achieves a Peak Performance of 1 ExaFlops (10^{18} FLOPs) under the 20 MWatts envelope
- Exascale computing will not just allow present solutions to run faster, but will enable new solutions not affordable with today's HPC technology

- 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

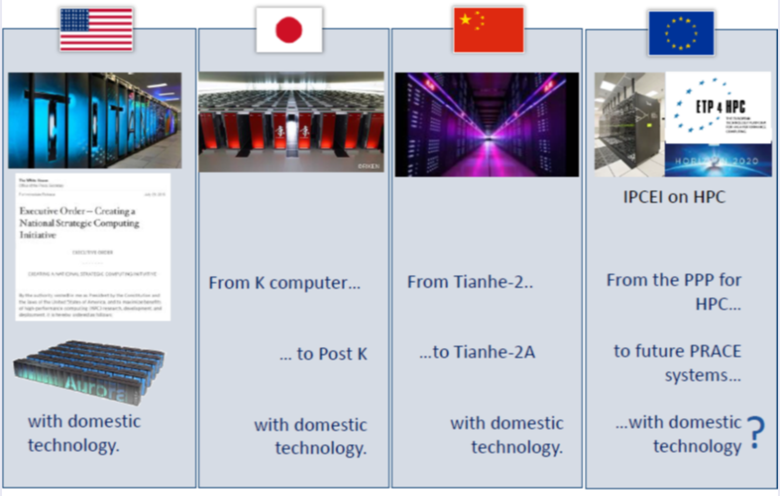


taken from Yelick's talks

MOTIVATION

THE INTERNATIONAL EXASCALE CHALLENGE

WORLDWIDE EXASCALE ROADMAPS



Executive Order – Creating a National Strategic Computing Initiative

with domestic technology.

From K computer...

... to Post K

with domestic technology.

From Tianhe-2..

...to Tianhe-2A

with domestic technology.

IPCEI on HPC

From the PPP for HPC...

to future PRACE systems...

...with domestic technology ?

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

MOTIVATION

THE INTERNATIONAL EXASCALE CHALLENGE

In addtion 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

MOTIVATION

THE THIRD IMPORTANT FACT

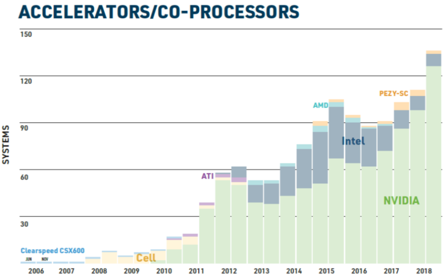
INTEL HIGH-END ARCHITECTURES

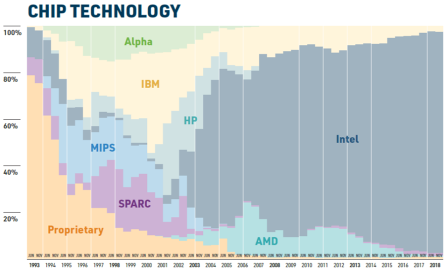
- Portability: Run x86 code
- General-purpose processors
- Latency-oriented vs. throughput-oriented processors
- Flexibility & *Cheaper* cost per unit

MOTIVATION

THE THIRD IMPORTANT FACT

Intel high-end architectures @ The Top500 list

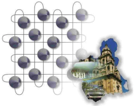

ACCELERATORS/CO-PROCESSORS


CHIP TECHNOLOGY

From the last list: November 2018

MOTIVATION
THE GACOP RESEARCH GROUP

The research of our group has been always centered around the topic of Supercomputing



MAIN RESEARCH LINES

- 1992 - 2002: Interconnection networks in HPC architectures
- 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

MOTIVATION

TALK OVERVIEW

- 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)

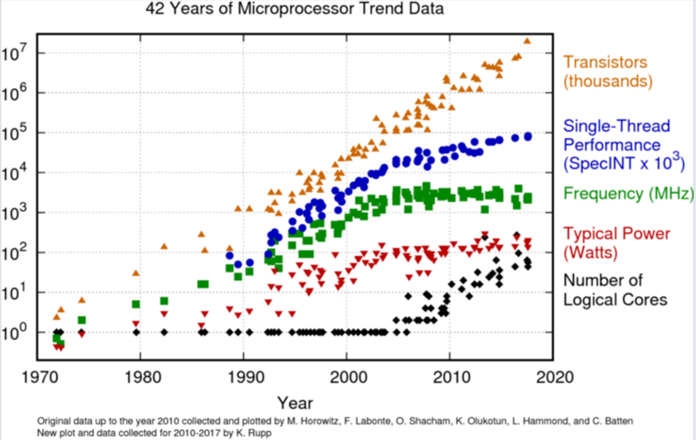
OUTLINE

1. BACKGROUND
2. CASE STUDY: 3-D STENCIL CODES
3. CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS
4. CASE STUDY: ACO
5. CONCLUSIONS AND FUTURE RESEARCH LINES

BACKGROUND

TECHNOLOGY TRENDS & PARALLEL PROGRAMMING

TECHNOLOGY EVOLUTION



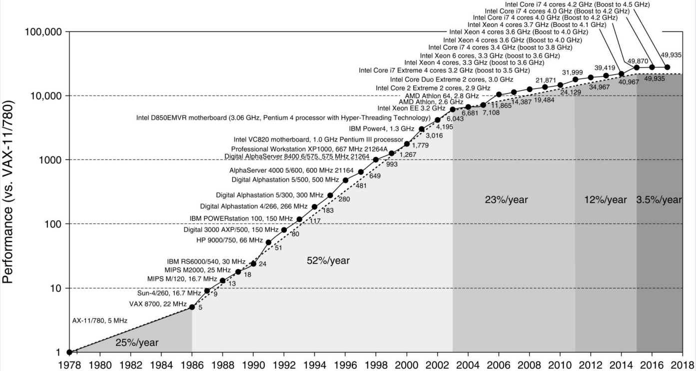
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

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

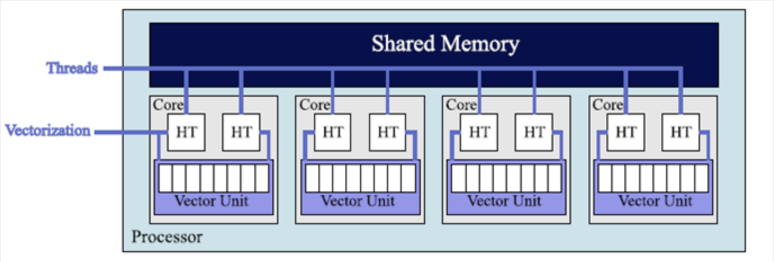
- Exploiting specialization: Accelerators
- Exploiting data parallelism (SIMD): Using wider vector instructions
- Exploiting thread parallelism (TLP): Using multi-cores

Hardware keeps evolving and Software must catch up!

BACKGROUND

TECHNOLOGY TRENDS & PARALLEL PROGRAMMING

A general multicore view



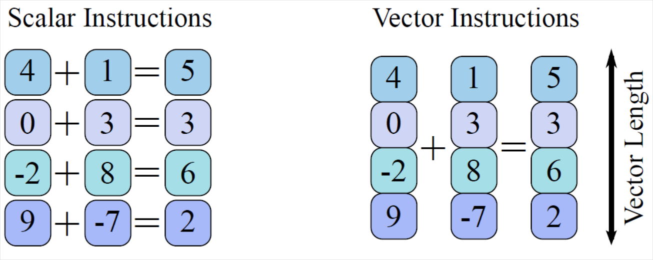
From Colfax HowTo Slides

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

BACKGROUND

TECHNOLOGY TRENDS & PARALLEL PROGRAMMING

Exploiting data parallelism (SIMD): Using vector instructions



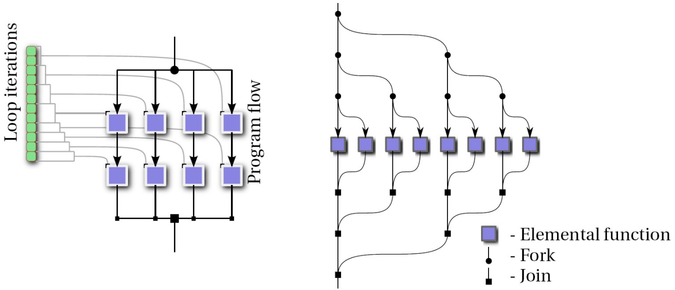
from Colfax HowTo slides

The wider the SIMD registers the better performance achieved using vectorization

BACKGROUND

TECHNOLOGY TRENDS & PARALLEL PROGRAMMING

Exploiting thread parallelism (TLP)



from Colfax HowTo slides

Threads are streams of instructions that share memory address space

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

BACKGROUND

TECHNOLOGY TRENDS & PARALLEL PROGRAMMING

We have used the OpenMP framework in this work

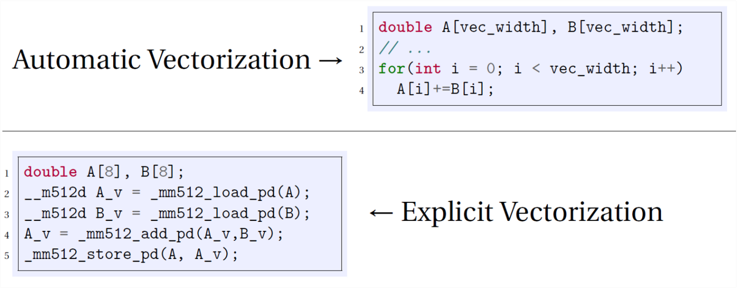
OPENMP MAIN FEATURES

- OpenMP = "Open Multi-Processing" = computing-oriented framework for shared-memory programming (`https://www.openmp.org/`)
- 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 →

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

```
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);
```

← Explicit Vectorization

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

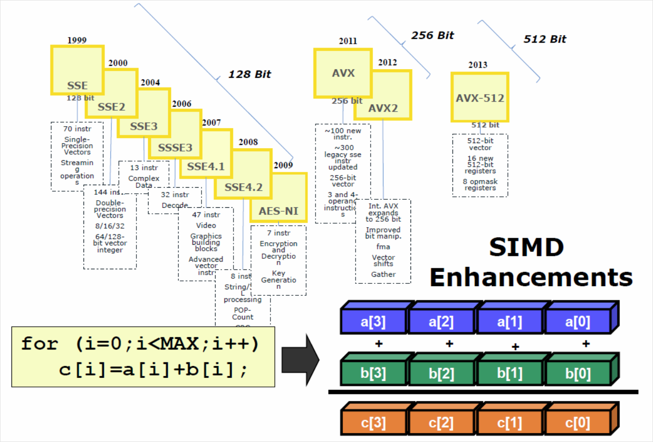
- Finding enough parallelism (Amdahl's Law)

All of these things make parallel programming even harder than sequential programming

BACKGROUND

INTEL HIGH-PERFORMANCE ARCHITECTURES

Evolution of vector parallelism on Intel Architectures



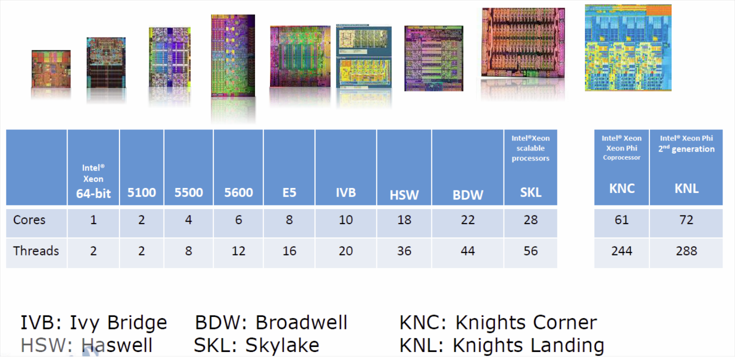
```
for (i=0;i<MAX;i++)
    c[i]=a[i]+b[i];
```

Source: Intel Developer Zone

BACKGROUND

INTEL HIGH-PERFORMANCE ARCHITECTURES

Evolution of thread parallelism on Intel high-end Architectures



	Intel® Xeon 64-bit	5100	5500	5600	E5	IVB	HSW	BDW	Intel®Xeon scalable processors SKL	Intel® Xeon Xeon Phi Coprocessor KNC	Intel® Xeon Phi 2nd generation KNL
Cores	1	2	4	6	8	10	18	22	28	61	72
Threads	2	2	8	12	16	20	36	44	56	244	288

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

Source: Intel Developer Zone

BACKGROUND

INTEL HIGH-PERFORMANCE ARCHITECTURES

Xeon family

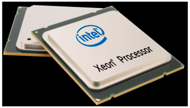
- General-purpose: Suitable for any workload
- 1-, 2-, 4-sockets: NUMA architecture
- Highly parallel
- Resource-rich
- Forgiving performance: High single-thread performance

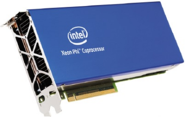
Xeon Phi (1st Gen - Knights Corner or `KNC`)

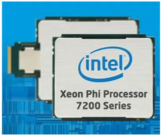
- Accelerator/coprocessor x86 based
- 61 in-order cores @ low frequency (1.2Ghz)
- Low single-thread performance
- High Memory Banwidth: 320 GB/sec
- Custom operating system on board

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)

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

BACKGROUND

OUR EVALUATION TEST BED

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

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

BACKGROUND

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

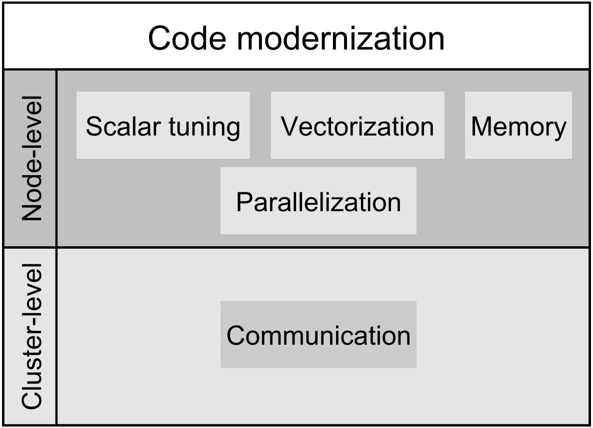
BRINGING CODES INTO THE PARALLEL AGE

- 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

BACKGROUND
CODE MODERNIZATION

OPTIMIZATION AREAS: NODE AND CLUSTER LEVEL



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

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

CASE STUDY: 3-D STENCIL CODES

PRESENTATION

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

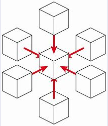
- Evaluate automatic vectorization (512-bit vector wide)
- Analyze thread parallelism: tuning its parameters
- Exploit memory locality: Cache blocking
- Target platforms: Xeon v2 (16 cores, 2 threads/core), KNC (61 cores, 4 threads/core), KNL (68 cores, 4 threads/core)

CASE STUDY: 3-D STENCIL CODES

BACKGROUND

3-D STENCIL CODES

- 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 neihgbours 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

```
for $time = 0; time < TimeMax; time++$ do
    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
```

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

CASE STUDY: 3-D STENCIL CODES

EVALUATION

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`.

CASE STUDY: 3-D STENCIL CODES

EVALUATION

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

Xeon v2	Threads					
	1	2	4	8	16	32
Acoustic	715	786	490	394	392	392
Seismic	2134	2494	1393	849	697	687
Heat	655	733	387	296	278	270

Xeon Phi KNC	Threads				
	1	61	128	183	244
Acoustic	7181	137	137	220	298
Seismic	36620	683	577	621	640
Heat	7953	145	96	146	189

(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

CASE STUDY: 3-D STENCIL CODES

CODE MODERNIZATION: VECTORIZATION (I)

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

CASE STUDY: 3-D STENCIL CODES

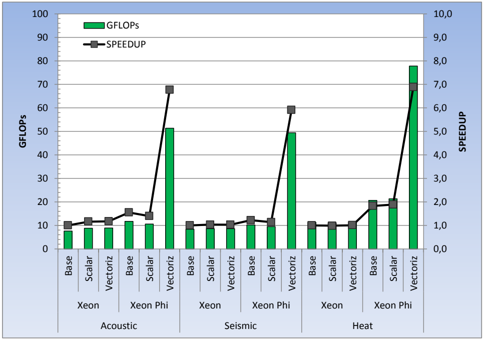
CODE MODERNIZATION: VECTORIZATION (II)

- *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*)

Version	Acoustic			Seismic			Heat		
	Xeon v2	KNC	KNL	Xeon v2	KNC	KNL	Xeon v2	KNC	KNL
Base	392	298	33	647	640	87*	270	189	30*
Vectorized	357	62	24	625	116	66	250	41	19

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

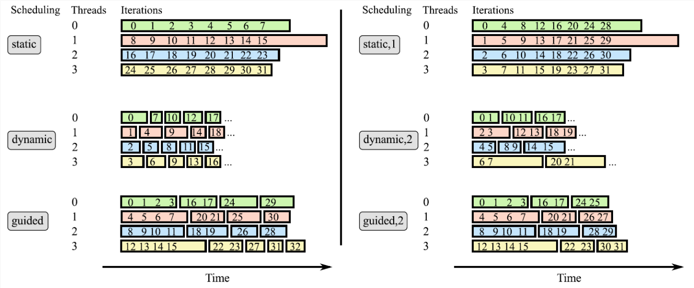
```
for $time = 0; time < TimeMax; time++$ do
    #pragma omp parallel for collapse (2);
    for $z = 1; z < depth - BorderSize; z++$ do
        for $y = 1; y < height - BorderSize; y++$ do
            for $x = 1; x < width - BorderSize; x++$ do
                $stencil\_solver\_kernel()$;
            end for
        end for
    end for
    $tmp = Input\_Grid; Input\_Grid = Output\_Grid; Output\_Grid = tmp$;
end for
```

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])`

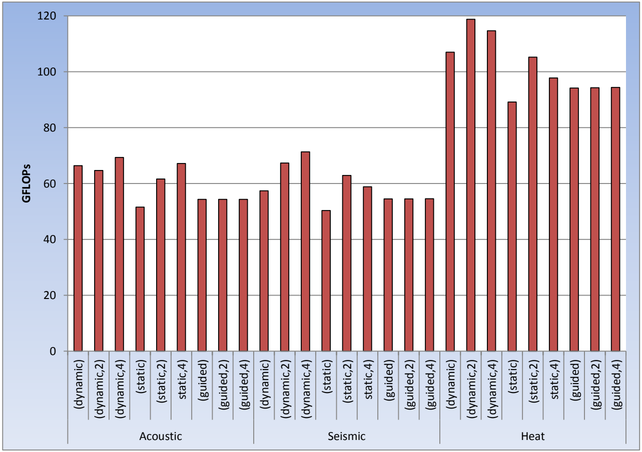


from Colfax HowTo slides

CASE STUDY: 3-D STENCIL CODES

CODE MODERNIZATION: EVALUATION OF THREAD PARALLELIZATION

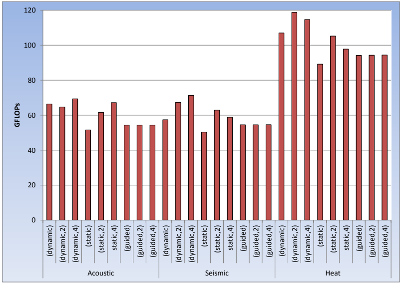
Evaluation of Scheduling policies on `KNC`



CASE STUDY: 3-D STENCIL CODES

CODE MODERNIZATION: EVALUATION OF THREAD PARALLELIZATION

Evaluation of Scheduling policies on `KNC`



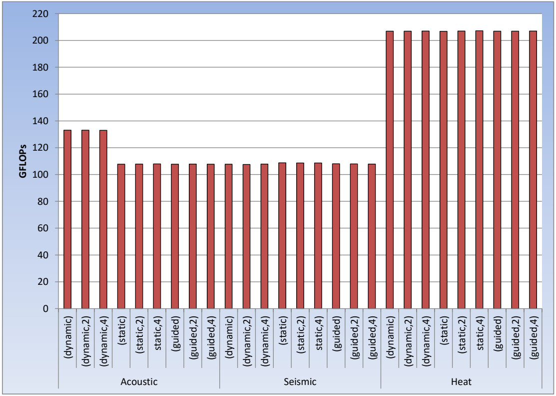
- `static` is not as good as supposed

  ⇒ 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

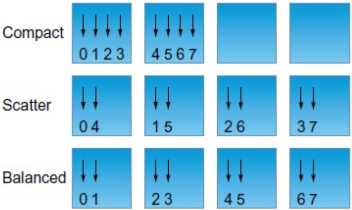
- For KNL, the variability between configurations is minimal
- `dynamic` was the best option for acoustic

CASE STUDY: 3-D STENCIL CODES

CODE MODERNIZATION: THREAD PARALLELIZATION (*affinity*)

Affinity Policies

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



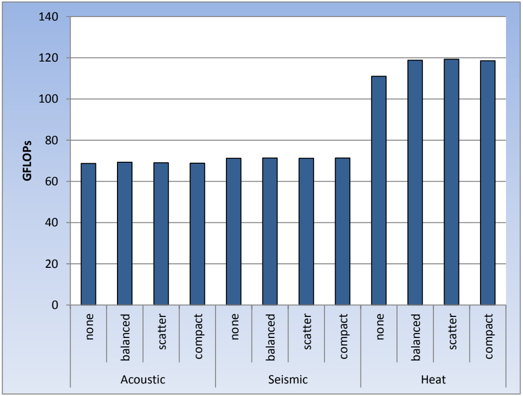
Compact	0 1 2 3	4 5 6 7		
Scatter	0 4	1 5	2 6	3 7
Balanced	0 1	2 3	4 5	6 7

- Setting affinity prevents thread migration
- Set up with the environment variable `KMP_AFFINITY`

CASE STUDY: 3-D STENCIL CODES

CODE MODERNIZATION: EVALUATION OF THREAD PARALLELIZATION

Evaluation of affinity policies on `KNC`

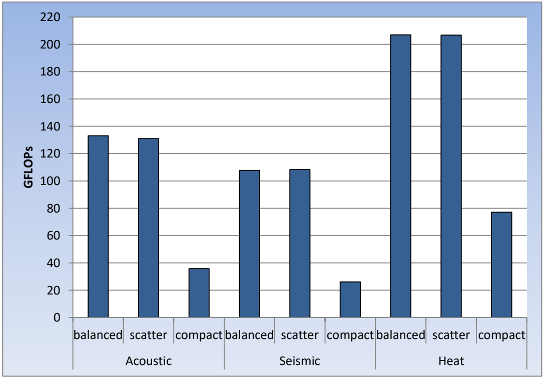


Affinity has a little influence on the execution time

CASE STUDY: 3-D STENCIL CODES

CODE MODERNIZATION: EVALUATION OF THREAD PARALLELIZATION

Evaluation of affinity policies on `KNL`



Cache mode with 64 threads

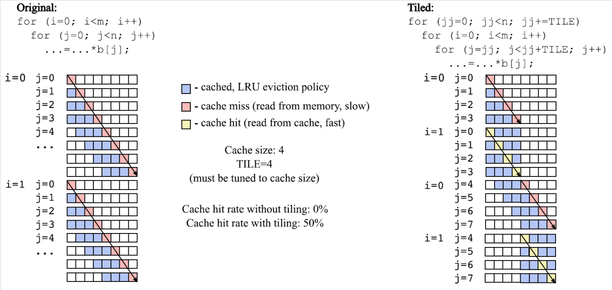
For KNL, the `compact` parameter is the worst option for all cases

CASE STUDY: 3-D STENCIL CODES

CODE MODERNIZATION: MEMORY OPTIMIZATION

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 *width_Tblock*, *height_Tblock* and *depth_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



Original:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=...*b[j];
```

Tiled:

```
for (jj=0; jj<n; jj+=TILE)
  for (i=0; i<m; i++)
    for (j=jj; j<jj+TILE; j++)
      ...=...*b[j];
```

🟦 - cached, LRU eviction policy

🟥 - cache miss (read from memory, slow)

🟨 - cache hit (read from cache, fast)

Cache size: 4
TILE=4
(must be tuned to cache size)

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%

from Colfax HowTo slides

CASE STUDY: 3-D STENCIL CODES

CODE MODERNIZATION: MEMORY OPTIMIZATION

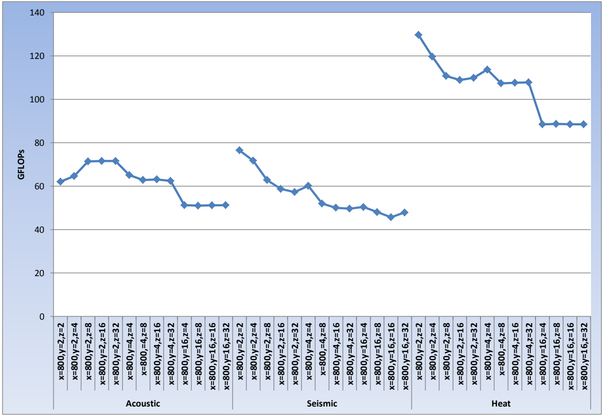
Blocking TECHNIQUE APPLIED TO THE 3-D *Stencil* SOLVER

```
for $bz = 1; bz < depth - BorderSize; bz+ = depth\_Tblock$ do
    for $by = 1; by < height - BorderSize; by+ = height\_Tblock$ do
        for $bx = 1; bx < width - BorderSize; bx+ = width\_Tblock$ do
            for $z = bz; z < MIN(bz + depth\_Tblock, depth - BorderSize); z++$ do
                for $y = by; y < MIN(by + height\_Tblock, height - BorderSize); y++$ do
                    for $x = bx; x < MIN(bx + width\_Tblock, width - BorderSize); x++$ do
                        $stencil\_solver\_kernel();$
                    end for
                end for
            end for
        end for
    end for
end for
```

CASE STUDY: 3-D STENCIL CODES

CODE MODERNIZATION: EVALUATION OF MEMORY OPTIMIZATION

Analyzing YZ blocking size - `KNC`



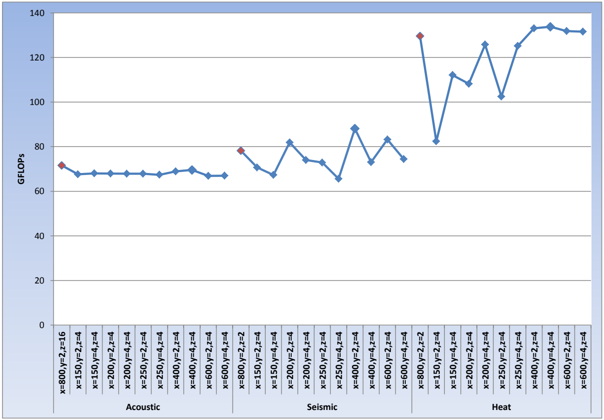
Blocking on the Y and Z axes

- Best block size: height_Tblock equals to 4, and depth_Tblock also equals to 4

CASE STUDY: 3-D STENCIL CODES

CODE MODERNIZATION: EVALUATION OF MEMORY OPTIMIZATION

Analyzing X blocking size - `KNC`



Blocking on the X axis

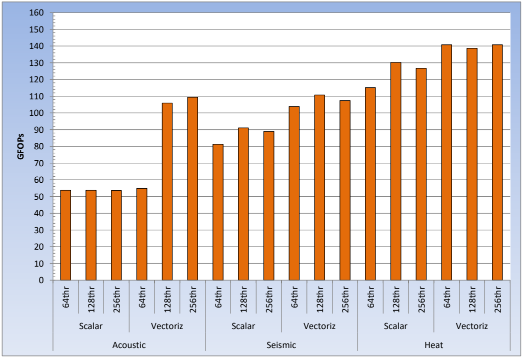
- Setting the block size to 4 (Y axis) and 4 (Z axis)
- Best block size: width_Tblock equals to 200

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



KNL in Cache mode

- 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)

Version	Acoustic			Seismic			Heat		
	Xeon v2	KNC	KNL	Xeon v2	KNC	KNL	Xeon v2	KNC	KNL
Base	8.7	10.4	81	7.6	8.4	81	10.2	20.1	129
+ Vectorized	8.8	51.2	112	7.6	49.1	108	10.3	84.5	204.3
+ Scheduling	-	69.7	138	-	69.1	112	-	119.5	204.3
+ Affinity	-	72.7	138	-	72.1	112	-	119.5	204.3
+ Blocking	-	76.2	110	-	84.1	108	-	138.3	140.3

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

CASE STUDY: 3-D STENCIL CODES

PAPERS PUBLISHED

- Mario Hernández, Baldomero Imbernón, Juan M. Navarro, José M. García, Juan M. Cebrián and José M. Cecilia. "Evaluation of the 3-D finite difference implementation of the acoustic diffusion equation model on massively parallel architectures". *Computers & Electrical Engineering* (ISSN: 0045-7906), Vol.: 46, pp. 190-201, 2015. Elsevier.

- Juan M. Cebrián, José M. Cecilia, Mario Hernández and José M. García. "Code Modernization Strategies to 3-D Stencil-based applications on Intel Xeon Phi: KNC and KNL". *Computers and Mathematics with Applications* (ISSN: 0898-1221), Vol.: 74, pp. 2557-2571, 2017. Elsevier B.V.

- Mario Hernández, Juan M. Cebrián, José M. Cecilia and José M. García. "Offloading strategies for Stencil kernels on the KNC Xeon Phi architecture: Accuracy versus Performance". *The International Journal of High Performance Computing Applications* (ISSN: 1741-2846), pp. 1-9, 2017. SAGE Publications. First Published November 7, 2017

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

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

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

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

PRESENTATION

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)

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

FOUNDATIONS

THE QUEST FOR ORTHOLOGS CONSORTIUM

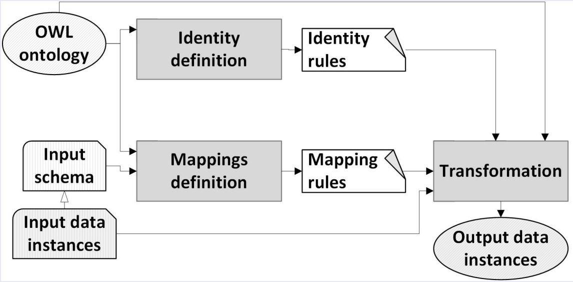
- The Quest for Orthologs (QfO) is a joint effort to improve and standardize orthology predictions through collaboration and the use of shared reference datasets
- ‘Orthology’ is the identification of gene relationships: Any of two or more homologous gene sequences found in different species related 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

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

FOUNDATIONS

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)



SWIT architecture

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

FOUNDATIONS

An input example (XML/relational database)

```
<species name="Escherichia-coli" NCBITaxId="83333">
<genes>
<gene id="1" protId="P07118" geneId="valS"/>
</genes>
</species>
<species name="Nematocida-parisii" NCBITaxId="881290">
<genes>
<gene id="2" protId="I3EQN8" geneId="NEPG_00863"/>
</genes>
</species>
<groups>
<orthologGroup id="1">
<geneRef id="1"/>
<geneRef id="2"/>
</orthologGroup>
</groups>
```

An output example (RDF/OWL dataset)

```
<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>
```

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

FOUNDATIONS

INPUT INSTANCES: ORTHOLOGY DATA

- 3 orthology databases have been used from `https://questfororthologs.org/orthology_databases`

- Inparanoid[a]: 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[b]: This resource stores groups of orthologs for several genomes. We have used the whole database, which is distributed in one 612 MB file.
- OMA[c]: 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/`

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS
FOUNDATIONS

OUTPUT DATA GENERATED (RDF TRIPLES)

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

The transformation of the 43 GB of the InParanoid database with identity rules required of 919 computational hours (around 38 days)!!!

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

CODE MODERNIZATION: REDESIGNING SWIT

HPC-SWIT BASICS

- 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

IDENTITY RULES

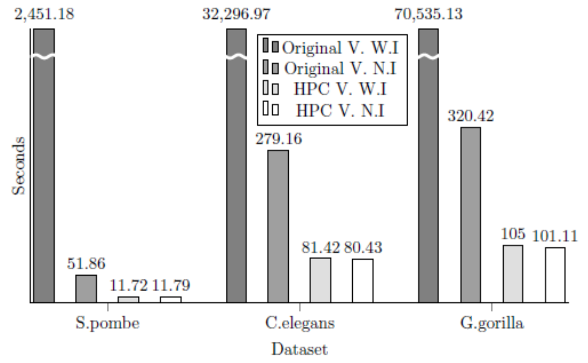
- The original version used SPARQL queries to detect redundant data, which is not efficient for large datasets or for identity rules with many conditions
- HPC-SWIT uses two new data structures: hash maps of vectors
- A hash map for *AND* conditions and another one for *OR* conditions
- The new method 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



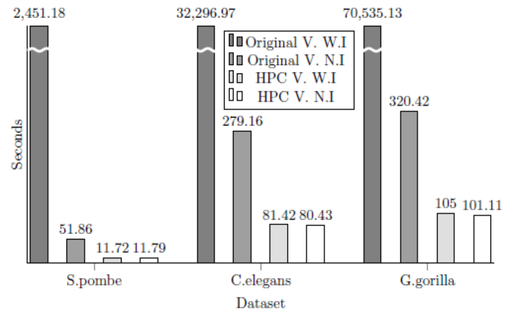
Original V. W.I and HPC V. W.I stands for the original version and optimized version of SWIT with identity rules, whilst Original V. and HPC V. N.I denotes the usage of these versions when no identity rules are applied

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



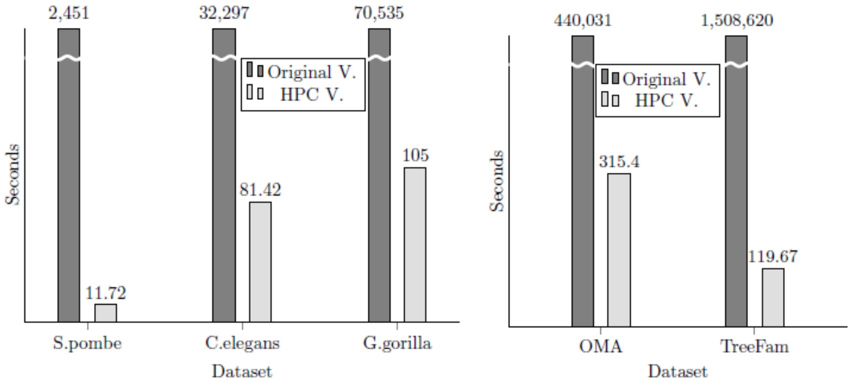
- A speedup between 3 and 4.5 is obtained without identity rules
- A speedup between 209 and 671 is obtained while using identity rules

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

CODE MODERNIZATION: REDESIGNING SWIT

Xeon v4
Using a single core (and one thread)

SINGLE CORE EXECUTION W/ IDENTITY RULES



CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

CODE MODERNIZATION: REDESIGNING SWIT

SINGLE CORE EXECUTION W/ IDENTITY RULES: SPEED-UP TABLE

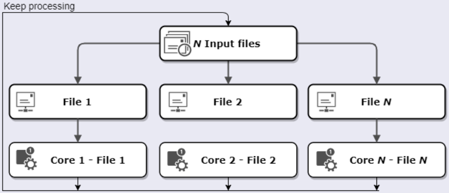
Dataset	HPC-SWIT vs SWIT Speed up
InParanoid *S.pombe*	209x
InParanoid *C.elegans*	396x
InParanoid *G.gorilla*	671x
InParanoid *WB*	240x
OMA	1, 395x
TreeFam	12, 606x

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

CODE MODERNIZATION: PARALLELIZATION

TASK PARALLELIZATION

- 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:

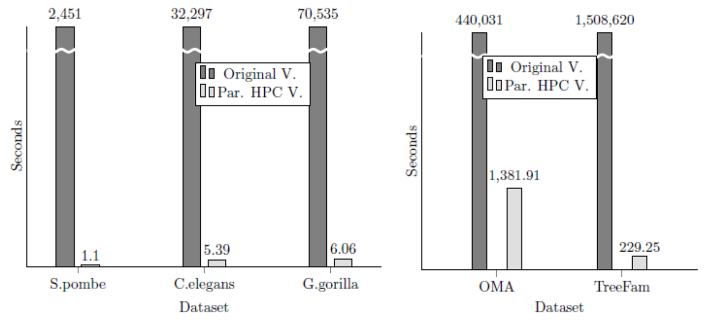
```
[user@ibsen ~]$ files = ( $( find $dir -maxdepth 1 -type f) )
[user@ibsen ~] parallel ./ swit {} " arguments " ::: ${ files [@]}
```

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

CODE MODERNIZATION: PARALLELIZATION

Xeon v4
40 cores (2 th/core)

EXECUTION TIME COMPARISON FROM ORIGINAL SWIT VS PARALLEL HPC-SWIT



- 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 (≈ 55 minutes)

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

CODE MODERNIZATION: PARALLELIZATION

PARALLEL HPC-SWIT VS SWIT SPEED UP

Database	Dataset	Sequential speed up	Parallel speed up
InParanoid	*S.pombe*	209x	1,964x
	C.elegans	397x	6,196x
	G.gorilla	672x	11,187x
	Whole database	240x	1,003x
OMA		1,395x	318x
TreeFam		12,606x	6,581x

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[a]

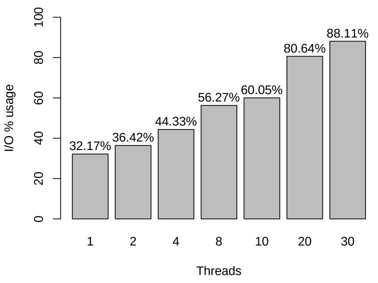
---

[a] 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

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

CODE MODERNIZATION: DISK USAGE

HPC-SWIT: I/O USAGE (%) FOR INPARANOID DATABASE



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)

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

CODE MODERNIZATION: DISK USAGE

COMPRESSION METHOD

- DEFLATE algorithm, standard *zip*
- A compression ratio of ≈ 27x for RDF/OWL files

HPC-SWIT: I/O EXECUTION TIME FOR INPARANOID DATABASE

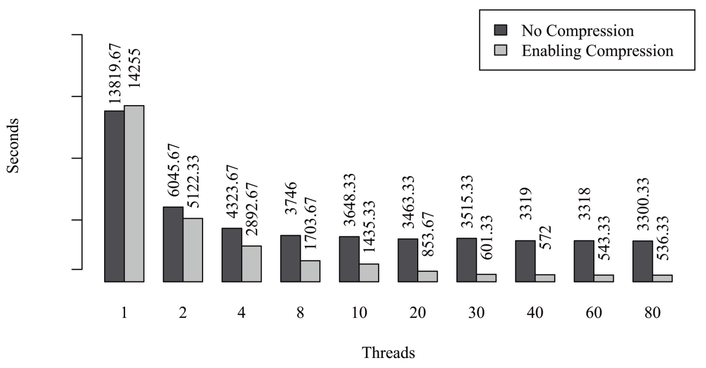
Version	Compression	Time (hrs)	Speed up
Original	No	919 (≈ 38 days)	Not applicable
HPC-SWIT	No	0.91 (≈ 55 m)	1, 003x
HPC-SWIT	Yes	0.14 (≈ 8 m 56 s)	6, 173x

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

CODE MODERNIZATION: DISK USAGE

Xeon v4
40 cores (2 th/core)

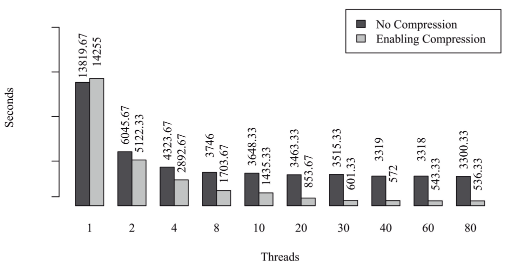
HPC-SWIT EXECUTION TIMES VARYING THE THREAD COUNT, W/ OR W/O COMPRESSION



CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

CODE MODERNIZATION: DISK USAGE

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.)

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS
CONCLUSIONS

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: $1,395$X (single core), worse in parallel
    - TreeFam database: $12,606$X (single core), worse in parallel

CASE STUDY: SEMANTIC WEB AND BIOINFORMATICS

PAPERS PUBLISHED

- José Antonio Bernabé-Díaz, María del Carmen Legaz-García, José M. García and Jesualdo Tomás Fernández-Breis. “Application of High Performance Computing Techniques to the Semantic Data Transformation”. In *Trends and Advances in Information Systems and Technologies*, Naples (Italy). pp. 691–700, 2018. Springer International Publishing. ISBN: 978-3-319-77703-0.

- José Antonio Bernabé-Díaz, María del Carmen Legaz-García, José M. García and Jesualdo Tomás Fernández-Breis. “Efficient, semantics-rich transformation and integration of large datasets”. *Expert Systems With Applications* (ISSN: 0957-4174). Volume 133, 1 November 2019, Pages 198-214.
doi: https://doi.org/10.1016/j.eswa.2019.05.010

OUTLINE

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

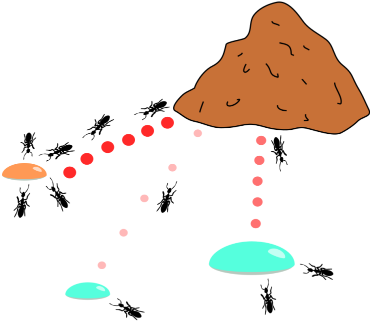
- The base code is based on the sequential Stützle's implementation in C++
- Best practices: scalar, parallelization, vectorization, and memory
- Evaluation of parallel efficiency (or scalability)
- Target platforms: Xeon v2 (16 cores, 2 threads/core), Xeon v4 (40 cores, 2 threads/core), KNC (61 cores,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

CASE STUDY: ACO

BACKGROUND

GENERAL STRUCTURE OF ACO ALGORITHMS

```
Initialization()
while not TerminationCondition() do
    TourConstruction()
    PheromoneUpdate()
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

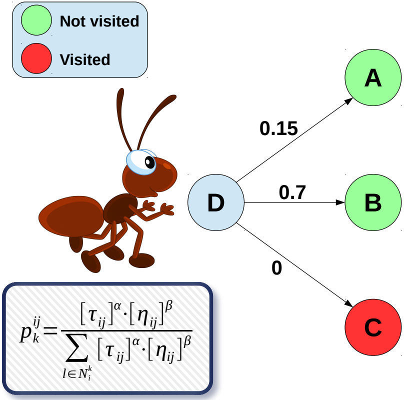
```
for $a = 1$ to $m$ do
    {Place ant on initial city}
    $initial\_city \leftarrow choose\_initial\_city()$
    $tour[a][1] \leftarrow initial\_city$
    $visited[a][initial\_city] \leftarrow$ true
    {Construct tour}
    for $step = 2$ to $n$ do
        $choose\_next(a, step)$
    end for
    $tour[a][n] \leftarrow tour[a][1]$
    $tour\_length[a] \leftarrow compute\_tour\_length(tour[a])$
end for
```

CASE STUDY: ACO

BACKGROUND

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
- τ_{ij} is the amount of pheromone associated with edge (i, j), $\eta_{ij} = 1/d_{ij}$ is a distance value computed *a priori*, α and β 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

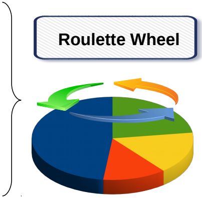
BACKGROUND

ROULETTE WHEEL SELECTION

```
Input: Ant identifier ($a$), construction step ($phase$).
$current\_city \leftarrow tour[a][phase - 1]$
{Selection Probabilities Computation}
$prob\_sum \leftarrow 0$
for $i = 1$ to $n$ do
    if $visited[a][i]$ then
        $prob[i] \leftarrow 0$
    else
        $prob[i] \leftarrow choice\_info[current\_city][i]$
        $prob\_sum \leftarrow prob\_sum + prob[i]$
    end if
end for
{City Selection}
$r \leftarrow random(0..prob\_sum)$
$city \leftarrow 1$
$partial\_sum \leftarrow prob[city]$
while $partial\_sum < r$ do
    $city \leftarrow city + 1$
    $partial\_sum \leftarrow partial\_sum + prob[city]$
end while
$tour[a][phase] \leftarrow city$
$visited[a][city] \leftarrow$ 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



CASE STUDY: ACO

EVALUATION

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 precomputed 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)

CASE STUDY: ACO

CODE MODERNIZATION: THREAD PARALLELIZATION

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

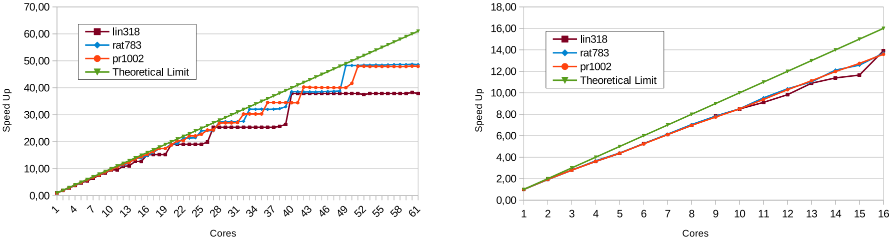
TOUR CONSTRUCTION

```
#pragma omp parallel for
for $a = 1$ to $m$ do
    $choose\_initial\_city(a)$
    for $step = 2$ to $n$ do
        $choose\_next\_city(a, step, thread\_id)$    → Selection function (99% of the time)
    end for
    $compute\_tour\_length(a)$
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)



Xeon Phi

Xeon v2

Parallel efficiency (or scalability) for tour construction

CASE STUDY: ACO

CODE MODERNIZATION: VECTORIZATION

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

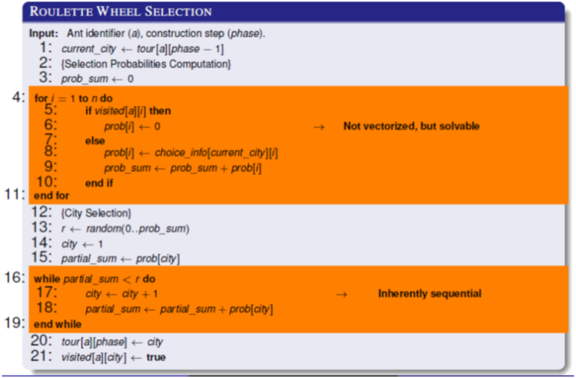
ROULETTE WHEEL SELECTION

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

```
$current\_city \leftarrow tour[a][phase-1]$
{Selection Probabilities Computation}
$prob\_sum \leftarrow 0$
for $i = 1$ to $n$ do
    if $visited[a][i]$ then
        $prob[i] \leftarrow 0$                          →  Not vectorized, but solvable
    else
        $prob[i] \leftarrow choice\_info[current\_city][i]$
        $prob\_sum \leftarrow prob\_sum + prob[i]$
    end if
end for
{City Selection}
$r \leftarrow random(0..prob\_sum)$
$city \leftarrow 1$
$partial\_sum \leftarrow prob[city]$
while $partial\_sum < r$ do
    $city \leftarrow city + 1$                           →  Inherently sequential
    $partial\_sum \leftarrow partial\_sum + prob[city]$
end while
$tour[a][phase] \leftarrow city$
$visited[a][city] \leftarrow$ true
```

CASE STUDY: ACO

CODE MODERNIZATION: VECTORIZATION



ROULETTE WHEEL SELECTION

```
Input: Ant identifier (a), construction step (phase).
current_city ← tour[a][phase − 1]
{Selection Probabilities Computation}
prob_sum ← 0
for i = 1 to n do
    if visited[a][i] then
        prob[i] ← 0
    else
        prob[i] ← choice_info[current_city][i]
        prob_sum ← prob_sum + prob[i]
    end if
end for
{City Selection}
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
tour[a][phase] ← city
visited[a][city] ← true
```

→ **Not vectorized, but solvable**

→ **Inherently sequential**

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

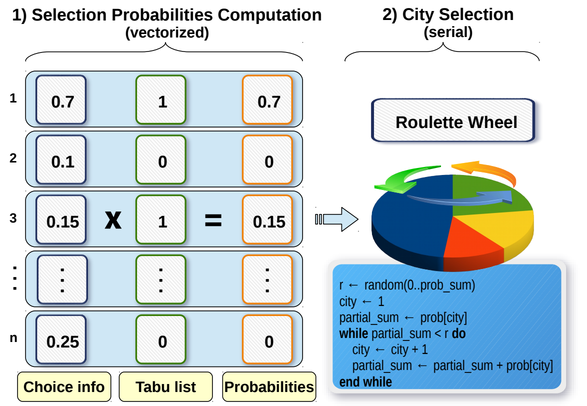
CASE STUDY: ACO

CODE MODERNIZATION: VECTORIZATION

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



```
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
```

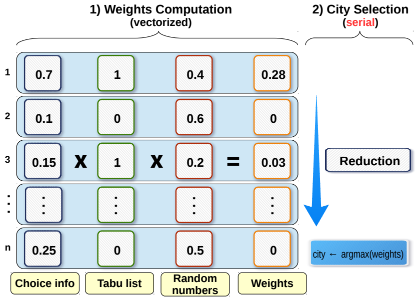
CASE STUDY: ACO

CODE MODERNIZATION: VECTORIZATION

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



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.

```
1:  $current\_city = tour[a][step - 1]$
2:  $city \leftarrow -1$
3:  $max\_weight \leftarrow -1$
    #pragma ivdep
4:  for $i = 1$ to $n$ do
5:      $w \leftarrow choice\_info[current\_city][i] * visited[a][i] * rand01(seeds[thread\_id][i])$
6:      if $w > max\_weigth$ then
7:          $city \leftarrow i$
8:          $max\_weight \leftarrow w$
9:      end if
10: end for
11: return $city$
```

VECTORIZATION REPORT (INTEL COMPILER)

Loop (lines 4-10):

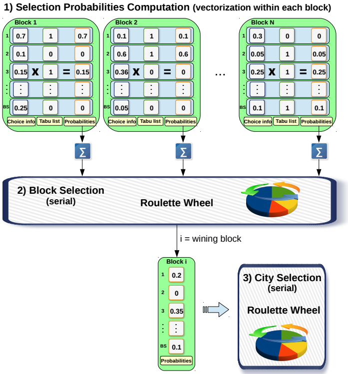
- Vectorized
- Unit stride
- Vector length = architecture's vector length

CASE STUDY: ACO

CODE MODERNIZATION: VECTORIZATION

ALTERNATIVE SELECTION FUNCTION: DS-ROULETTE

- 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)

Roulette Wheel

i = wining block

3) City Selection (serial)

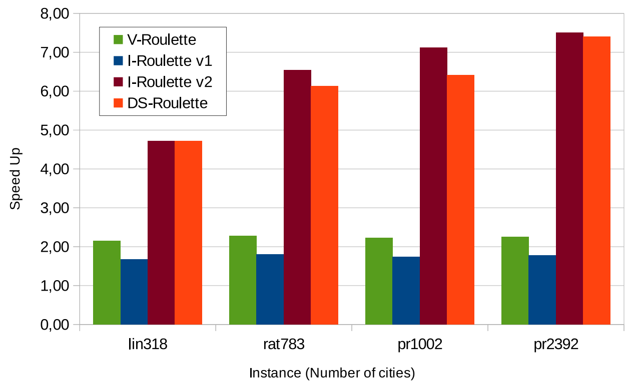
Roulette Wheel

CASE STUDY: ACO

CODE MODERNIZATION: VECTORIZATION

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

SELECTION FUNCTIONS ON XEON PHI KNC



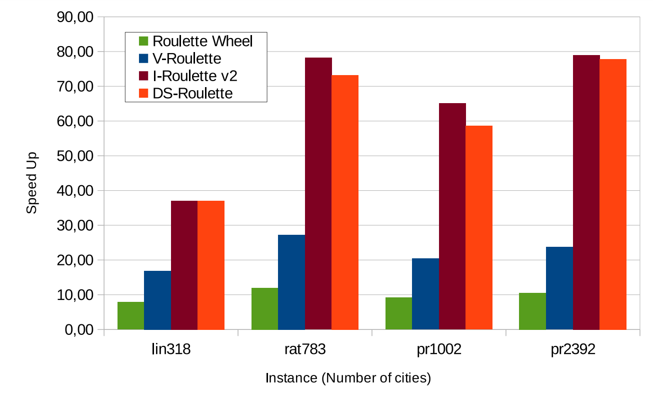
Speed up for the tour construction stage with different selection functions (compared to Roulette Wheel

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

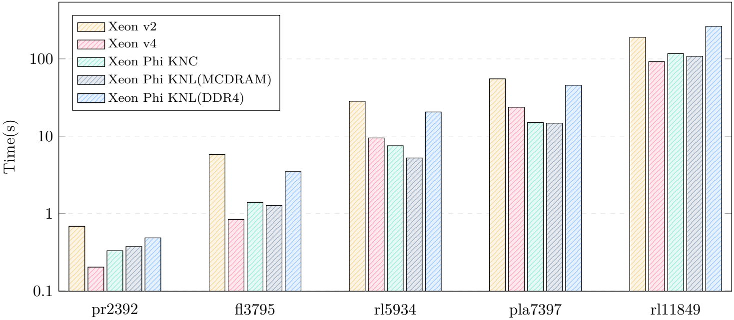


Execution time on Xeon Phi KNC compared to sequential code on Xeon v2

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)

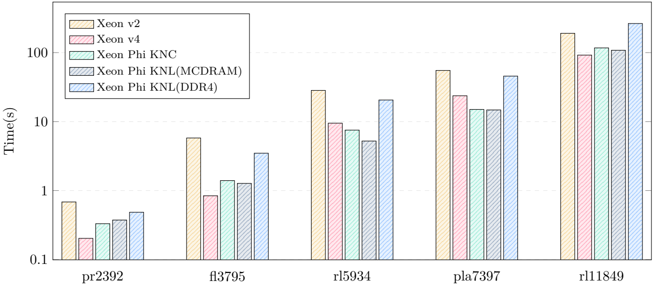


Execution time (s) for tour construction on different architectures

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)

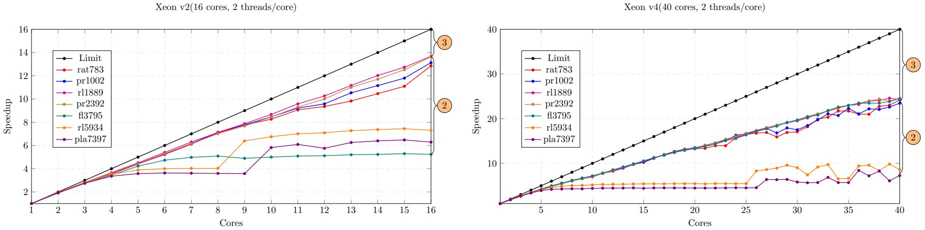


- Intel Xeon v4 outperforms the other architectures when it runs instances of up to 3795 cities (speedup of 10X over Xeon v2)
- KNC and KNL outperform Xeon v4 for larger instances
- Speedups for KNC up to 6X and for KNL(MCDRAM) up to 9X
- For the largest instance Xeon v4 is slightly better than the two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

CASE STUDY: ACO

CODE MODERNIZATION: THREAD PARALLELIZATION

Parallel Efficiency on Xeon multicore

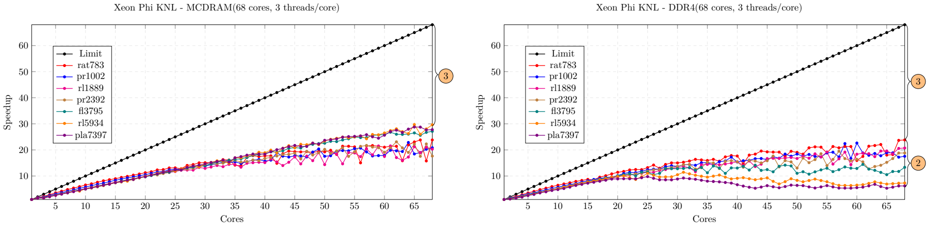


- Xeon v2 obtains good parallel efficiency for small and medium-sized problem instances (around 80%), but it decreases for larger problem sizes (around 40%)
- Xeon v4 shows worse scalability, ranging from 62% to only 20% for large problem sizes

CASE STUDY: ACO

CODE MODERNIZATION: THREAD PARALLELIZATION

Parallel Efficiency on Xeon Phi KNL

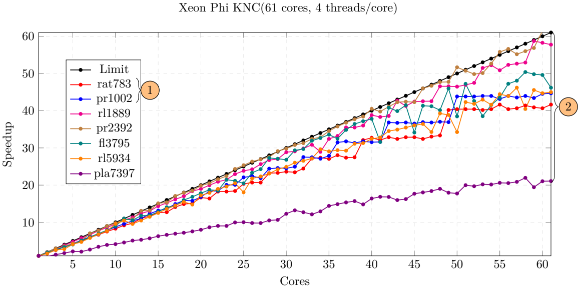


- Xeon Phi KNL with MCDRAM memory achieves a parallel efficiency ratio from 31% to 20%
- Xeon Phi KNL with DDR4 memory achieves a parallel efficiency ratio from 30% to 8%

CASE STUDY: ACO

CODE MODERNIZATION: THREAD PARALLELIZATION

Parallel Efficiency on Xeon Phi KNC



- 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

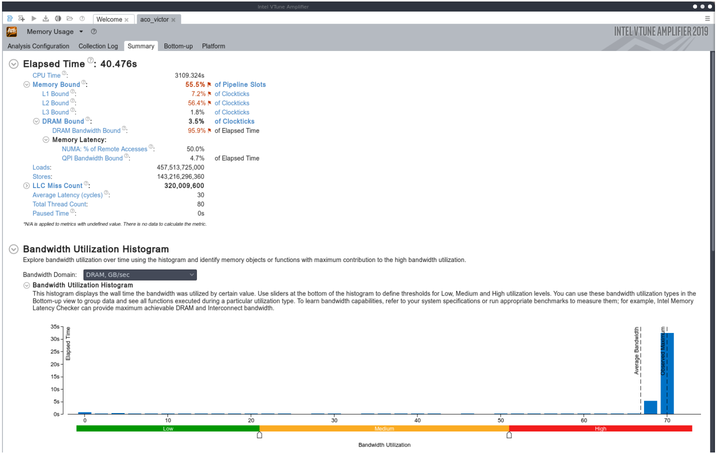
PARALLEL EFFICIENCY: FEASIBLE EXPLANATIONS

1. Core load unbalance: Limited impact (depending of number of cores and problem size)
2. Memory bandwidth limitations: The key factor
3. NUMA effects on data placement: Not for KNC

CASE STUDY: ACO

CODE MODERNIZATION: THREAD PARALLELIZATION

Memory bandwidth analysis with Vtune



```
Elapsed Time: 40.476s
  CPU Time:                                   3109.324s
  Memory Bound:                               55.5%  of Pipeline Slots
    L1 Bound:                                  7.2%  of Clockticks
    L2 Bound:                                 56.4%  of Clockticks
    L3 Bound:                                  1.8%  of Clockticks
    DRAM Bound:                                3.5%  of Clockticks
      DRAM Bandwidth Bound:                   95.9%  of Elapsed Time
      Memory Latency:
        NUMA: % of Remote Accesses:           50.0%
        QPI Bandwidth Bound:                   4.7%  of Elapsed Time
  Loads:                            457,513,725,000
  Stores:                           143,216,296,360
  LLC Miss Count:                       320,009,600
  Average Latency (cycles):                      30
  Total Thread Count:                            80
  Paused Time:                                   0s
*N/A is applied to metrics with undefined value. There is no data to calculate the metric.
```

Bandwidth Utilization Histogram

Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth utilization.

Bandwidth Domain: DRAM, GB/sec

Bandwidth Utilization Histogram

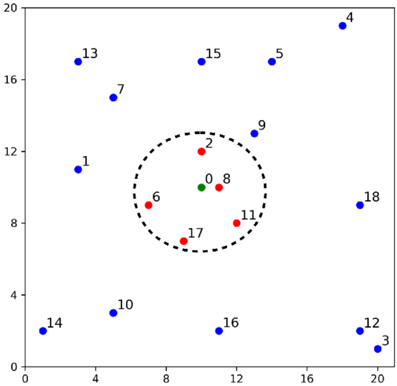
This histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and Interconnect bandwidth.

- 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

CASE STUDY: ACO

CODE MODERNIZATION: THREAD PARALLELIZATION

Recent proposal: Using a K-nearest neighbor list

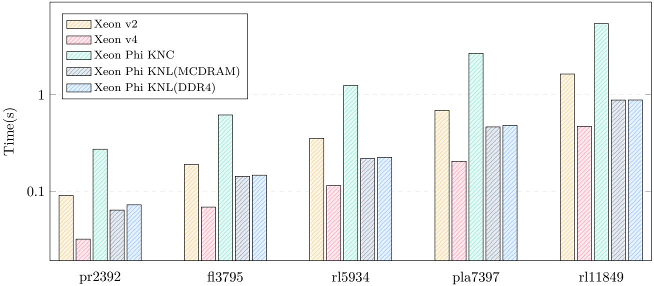


- K-nearest neighbor criteria: neighbors ordered by `choice_info` values
- We tested several values for K
- Always multiples of 8 ⇒ Final value: groups of 32 neighbours

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

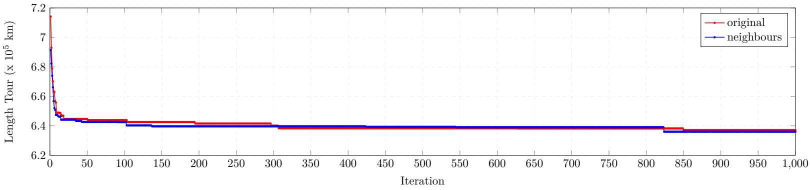
- A huge reduction time: KNC around 10X and the rest over 100X
- As memory bandwidth problems were solved:
    - Xeon v4 was again the best architecture
    - KNL's execution time is the same with DDR4 and MCDRAM

CASE STUDY: ACO

ON-GOING WORK: K-NEAREST NEIGHBOR LIST

Quality of Solution

Independent of the target platform



Quality of solution measured as tour length for 1000 iterations

CASE STUDY: ACO

CONCLUSIONS

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

CASE STUDY: ACO

PAPERS PUBLISHED

- José M. Cecilia, José M. García, Andy Nisbet, Martyn Amos and Manuel Ujaldón. “Enhancing Data Parallelism for Ant Colony Optimisation on GPUs”. *Journal of Parallel and Distributed Computing* (ISSN: 0743-7315), Vol.: 73, pp. 42-51, 2013. Elsevier.

- Antonio Llanes, José M. Cecilia, Antonia Sánchez, José M. García, Martyn Amos and Manuel Ujaldón. “Dynamic load balancing on heterogeneous clusters for parallel ant colony optimization”. *Cluster Computing* (ISSN: 1573-7543), Vol.: 19, pp. 1-11, 2016. Springer International Publishing.

- José M. Cecilia and José M. García. “Re-engineering the ant colony optimization for CMP architectures”. *The Journal of Supercomputing* (ISSN: 0920-8542), pp 1–22, First Online: 30 April 2019. https://doi.org/10.1007/s11227-019-02869-8

OUTLINE

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

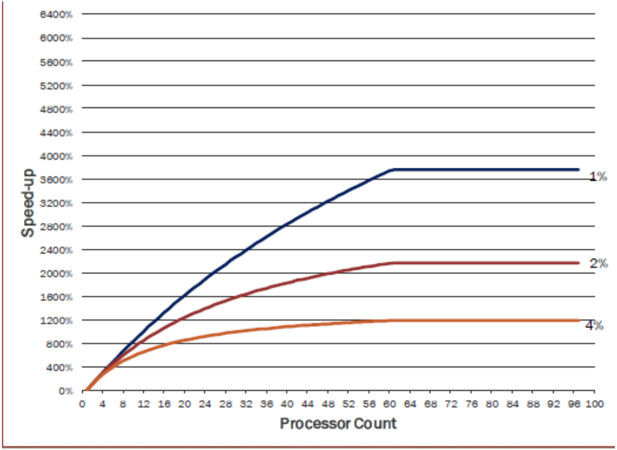


FIGURE: Speedup versus % "Serial" Processing Time; from Hennessy's talk [2018]

CONCLUSIONS & FUTURE WORK

DSLS & DSAS

CHALLENGES AHEAD

- Application focus shifts: From desktop to individual, mobile devices and ultrascale cloud computing, IoT, Bid 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

VNIVERSITAS · STVDIORVM · MVRCIANA
ANNO · MCCLXXII

CREDITS

FUNDING SOURCES

- 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

CREDITS

INSPIRATION SOURCES

- HOW Series “Deep Dive”: Webinars on Performance Optimization (2017 Edition). Colfax Research (Colfax International). 2017.
- “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
- Intel Xeon Phi Processor High Performance Programming, Knights Landing Edition. James Reinders, Jim Jeffers and Avinash Sodani, Morgan Kaufmann, 2016. (ISBN 978-0-12-809194-4)

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