From Multicores to Manycores Processors: Challenging Programing Issues with the MPPA/Kalray

Márcio Castro[1], Emilio Francesquini[2,3], Thomas Messi Nguélé[4] and Jean-François Mehaut[2,5]

[1] Federal University of Rio Grande do Sul
[2] University of Grenoble
[3] University of São Paulo
[4] University of Yaoundé
[5] CEA - DRT

JLPC Workshop - November 2013

Introduction

Demand for higher processor performance

- Increase of **clock frequency**
- Turning point: **power consumption** changed the course of development of new processors

Trend in parallel computing

- The number of cores per die **continues to increase**
- Hundreds or even thousands of cores

Different execution and programming models

- **Light-weight manycore processors**: autonomous cores, POSIX threads, data and task parallelism
- **GPUs**: SIMD model, CUDA and OpenCL

Introduction

Energy efficiency is already a primary concern

- **Mont-Blanc project**[1]: develop a full energy-efficient HPC system using low-power commercially available embedded processors

What we've been seeing

- Performance and energy efficiency of **numerical kernels on multicores**

What is missing?

1. Few works on **embedded** and **manycore** processors
2. What about **irregular applications**?

[1] http://montblanc-project.eu

Introduction

Our goals

- Analyze the **computing** and **energy** performance of multicore and manycore processors
- Consider an **irregular** application as a case study: **Traveling-Salesman Problem (TSP)**
- Consider a **new manycore chip** (MPPA-256) and other **general-purpose** (Intel Sandy Bridge) and **embedded** (ARM) multicore processors

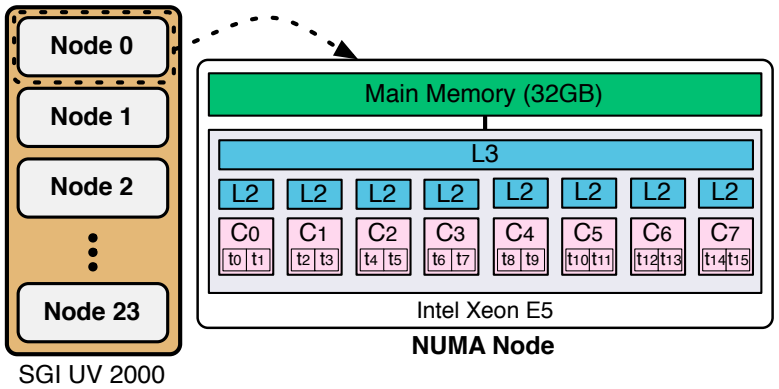
Platforms

We considered 4 platforms in this study

- **General-purpose** and embedded processors

General-purpose processors

- **Xeon E5**: Intel Xeon E5-4640 Sandy Bridge-EP processor chip, which has 8 CPU cores (16 threads with Hyper-Threading support enabled) running at 2.40GHz
- **Altix UV 2000**: NUMA platform composed of 24 Xeon E5 processors interconnected by NUMA-link6



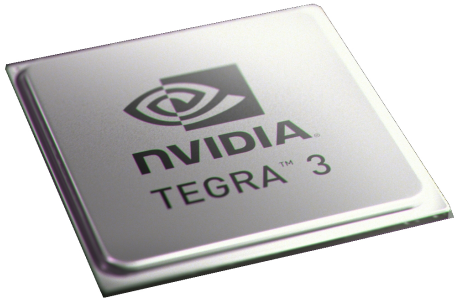
Platforms

We considered 4 platforms in this study

- General-purpose and **embedded processors**

Embedded processors

- **Carma**: a development kit from SECO that features a quad-core Nvidia Tegra 3 running at 1.3GHz



- **MPPA-256**: a single-chip manycore processor developed by Kalray that integrates 256 user cores and 32 system cores running at 400MHz

Kalray

French Startup founded in 2008

- Headquarters in Grenoble (Montbonnot), Paris (Orsay)
- Offices in Japan and US (California)

Disruptive Technology

- Based on research from CEA-LETI (Hardware), INRIA (compilers), CEA-LIST (data-flow languages)
- Multi-Purpose, Massively Parallel, Low Power Processors and System Software

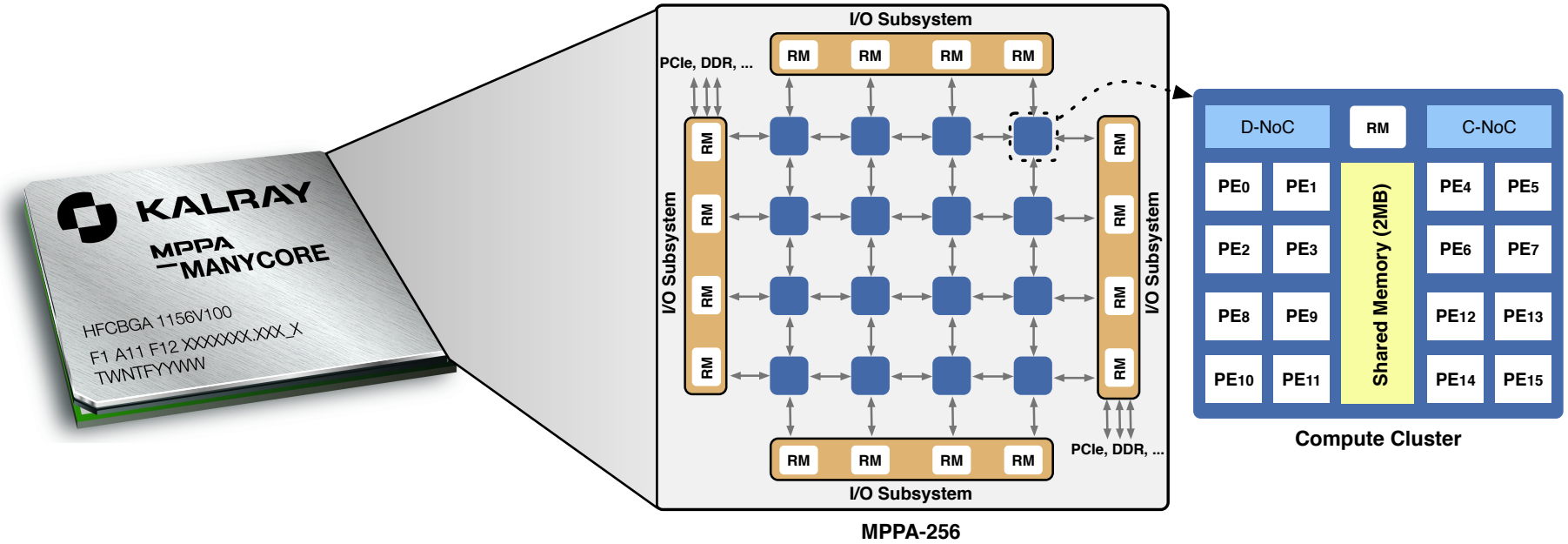
People

- 55 employees, out of which 45 in Research and Development

Fabless company

- Manufacturing is delegated to state of the art foundries (TSMC, 28 nm)

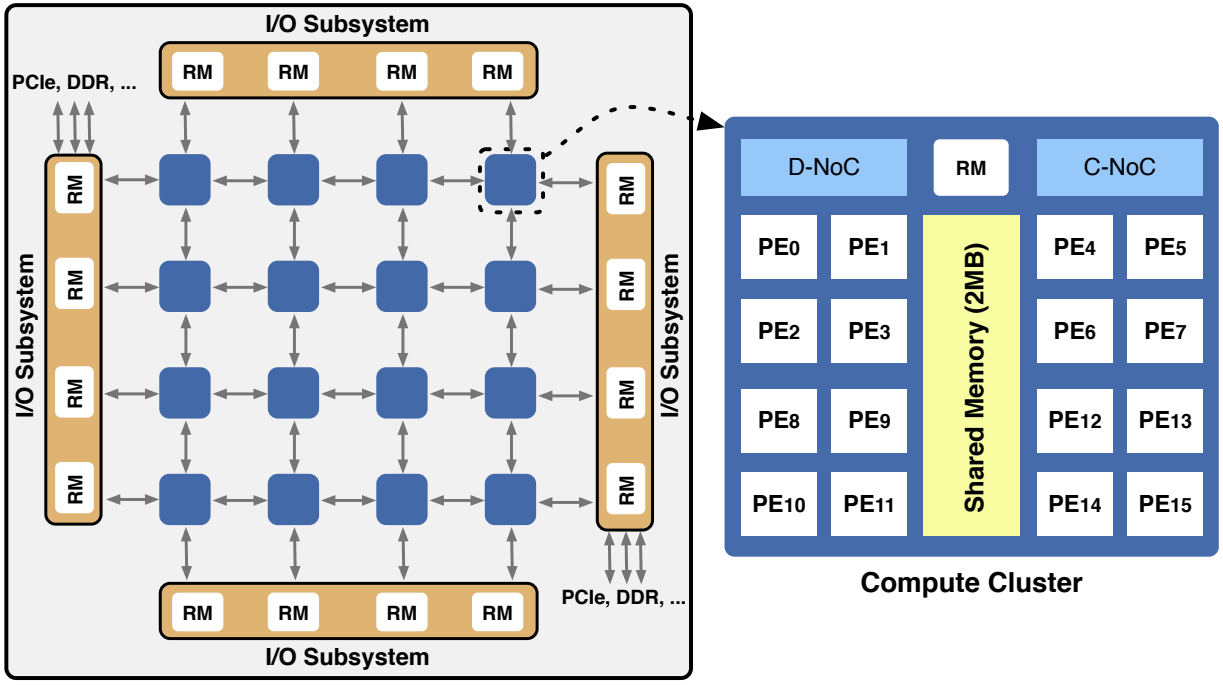
Kalray MPPA-256



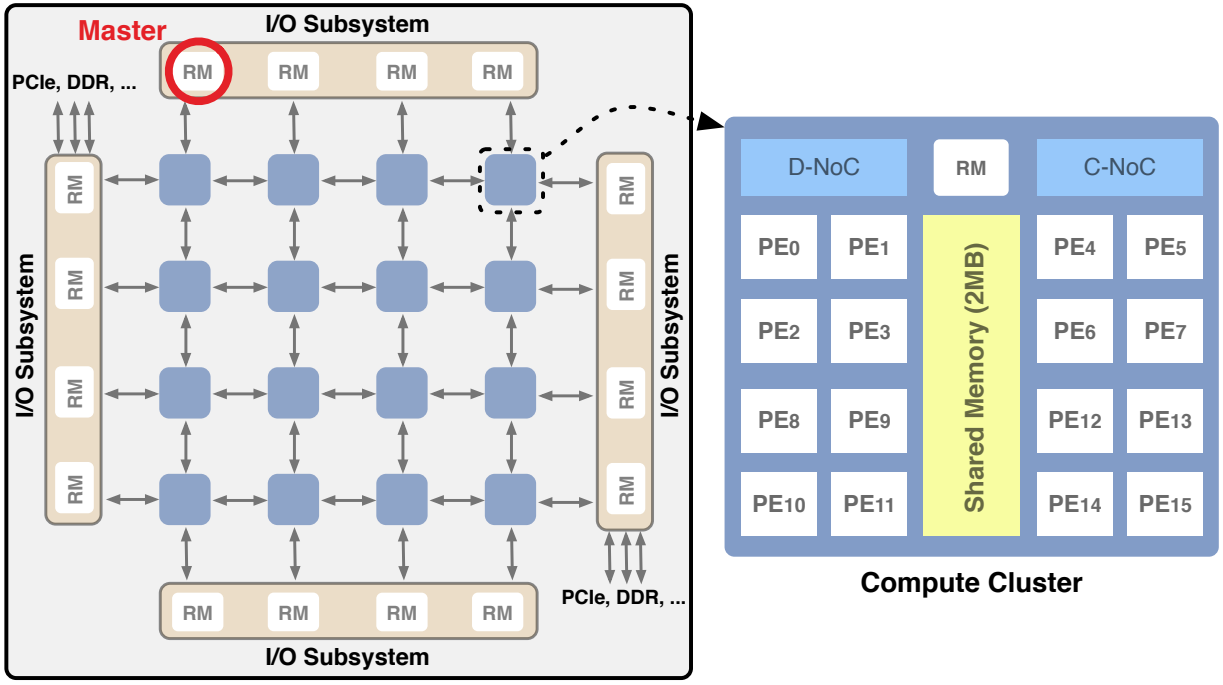
Inside the chip:

- 256 cores (400MHz): **16 clusters – 16 PEs per cluster**
- PEs **share 2MB of memory**
- **Absence of cache coherence** protocol inside clusters
- Communication between clusters: **Network-on-Chip** (NoC)
- **4 I/O subsystems**: 2 of them connected to external memory

Kalray MPPA-256



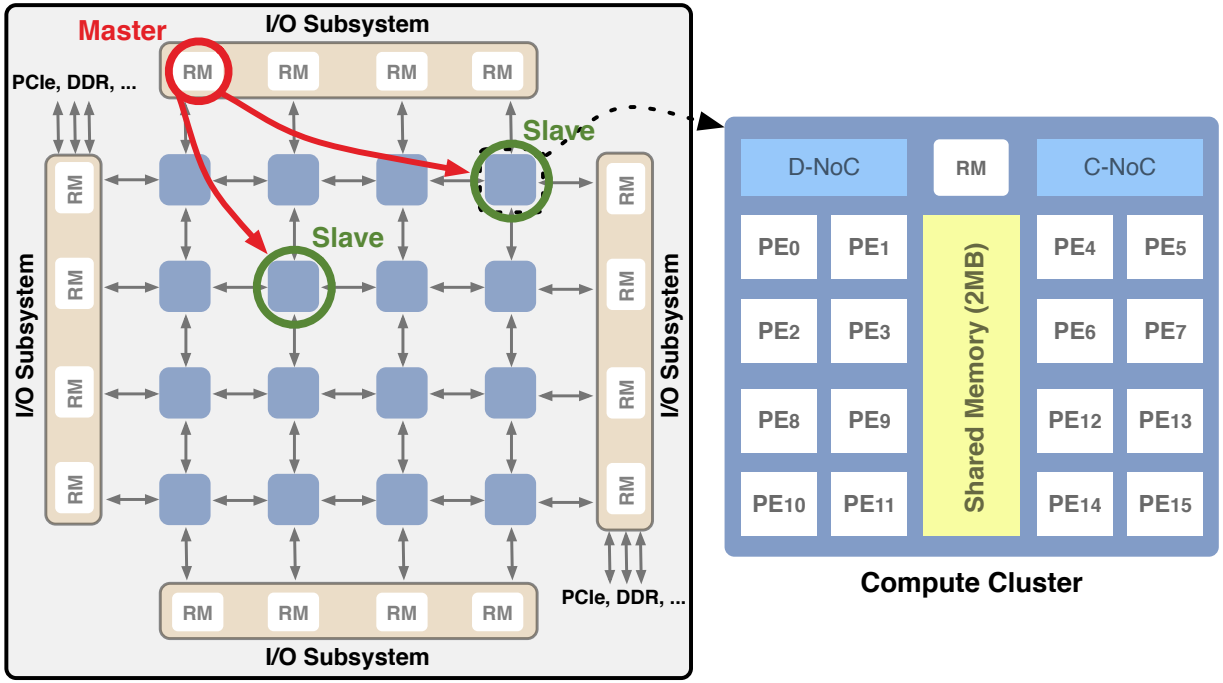
Kalray MPPA-256



MPPA-256

- A **master** process runs on an **RM** of an **I/O subsystem**

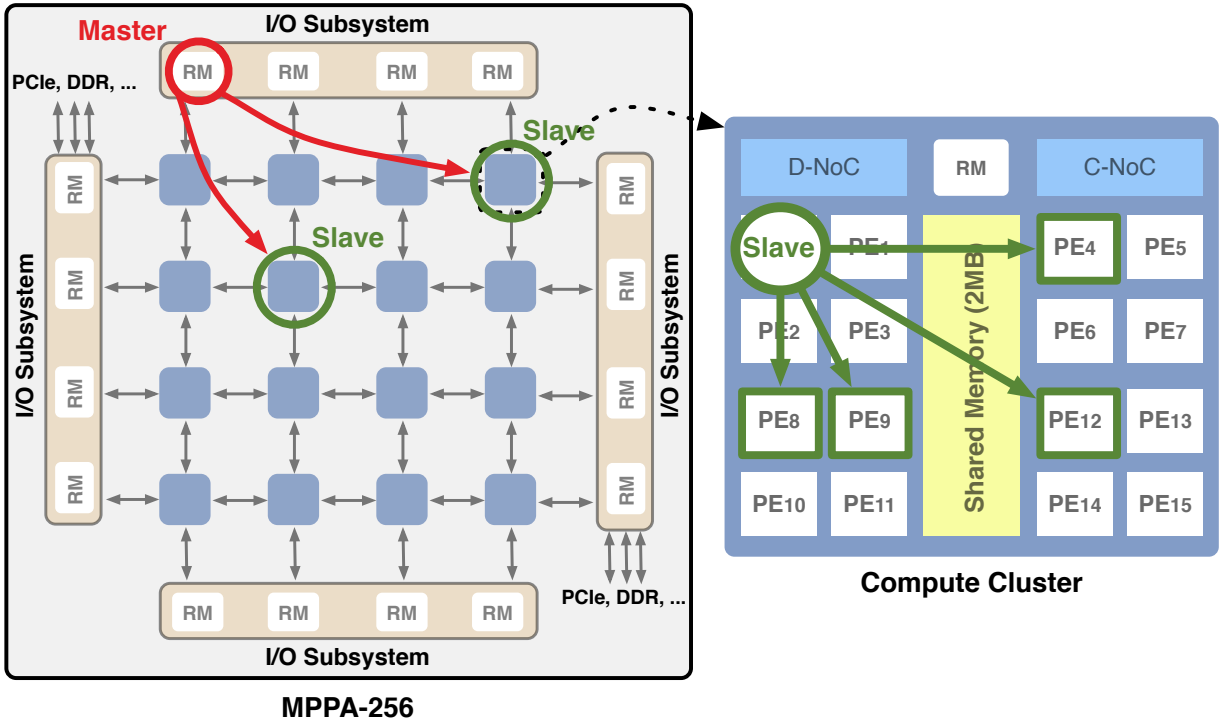
Kalray MPPA-256



MPPA-256

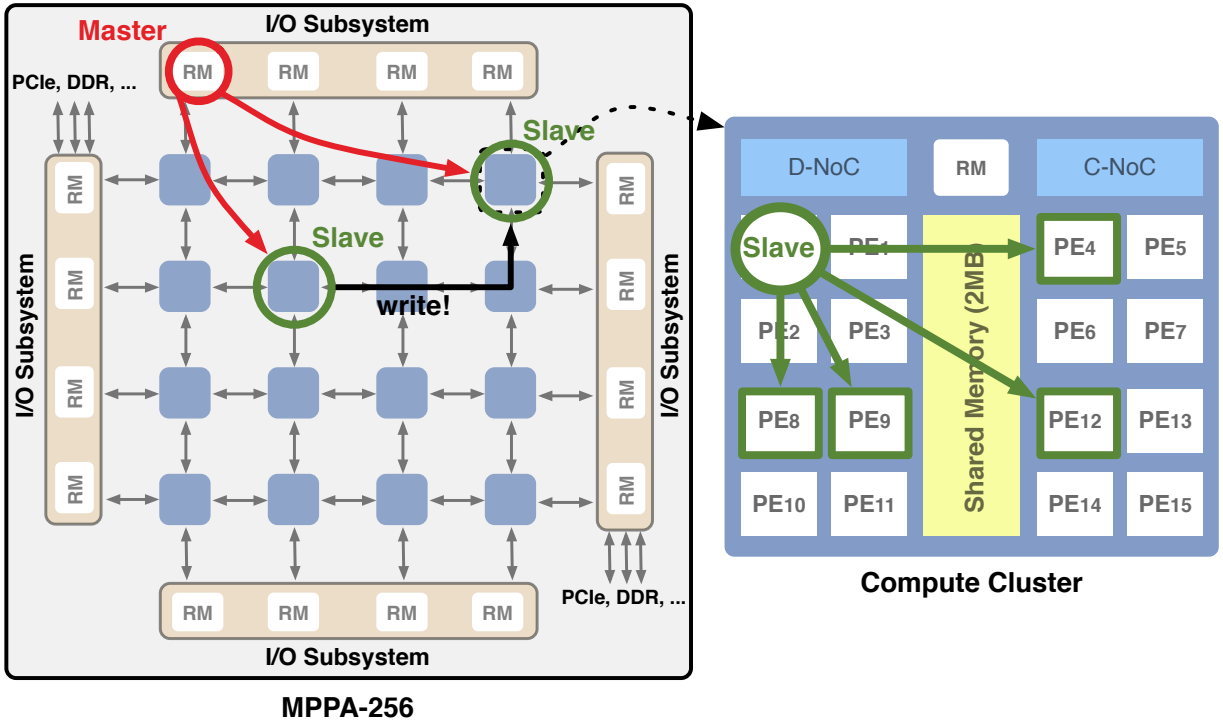
- The **master process** spawns **slave processes**
- 1 slave process per cluster

Kalray MPPA-256



- The **slave process** runs on **PE0** and may create up to 15 threads ➡ one for each PE
- **Threads share 2MB** of memory within the cluster

Kalray MPPA-256



- Communications: **remote writes**
- **Data travel through the NoC**

Case study: Travelling salesman problem (TSP)

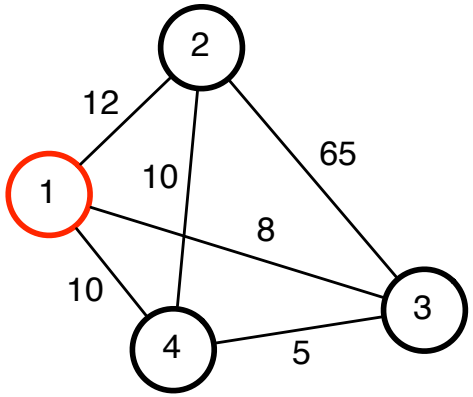
Definition

- It consists of finding a **shortest possible path** that passes through n cities, **visiting each city only once**, and returns to the city of origin.

Representation

- Complete undirected graph
- **Nodes**: cities
- **Edges**: distances (costs)

Example: $n = 4$
Possible paths from 1:
1-2-3-4-1, 1-2-4-3-1,
1-3-2-4-1, 1-3-4-2-1, ...



TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one

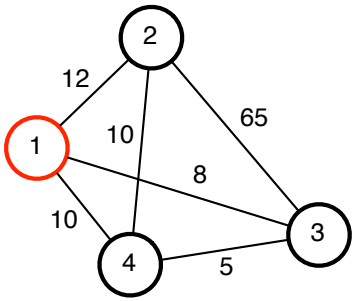
```
global min_path
procedure TSP_SOLVE(last_city, current_cost, cities)
 if cities = ∅
   then return (current_cost)
 for each i ∈ cities
   do
   { new_cost ← current_cost + costs[last_city, i]
   { if new_cost < min_path
   {   then { new_min ← TSP_SOLVE(i, new_cost, cities\{i})
   {        { ATOMIC_UPDATE_IF_LESS(min_path, new_min)

main
 min_path ← ∞
 TSP_SOLVE(1, 0, {2, 3, ..., n_cities})
 output (min_path)
```

TSP: Sequential algorithm

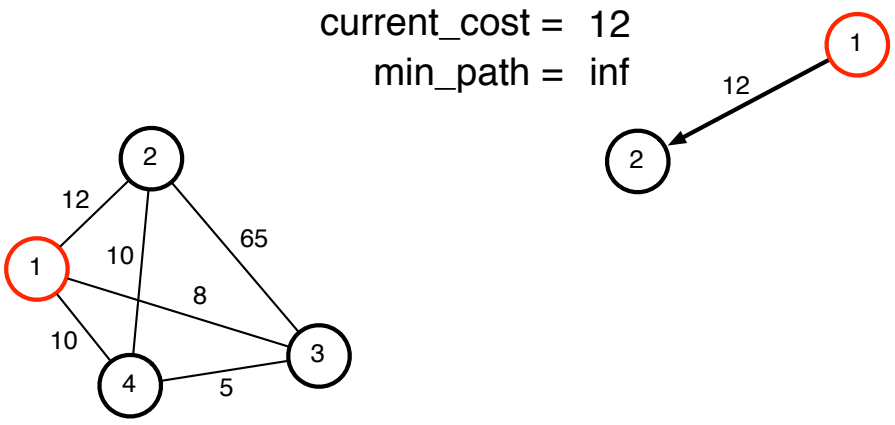
- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one

current_cost = 0
min_path = inf



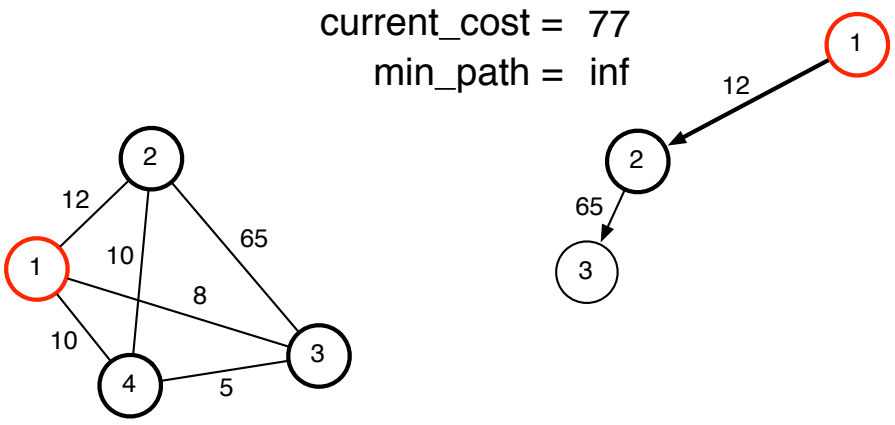
TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one



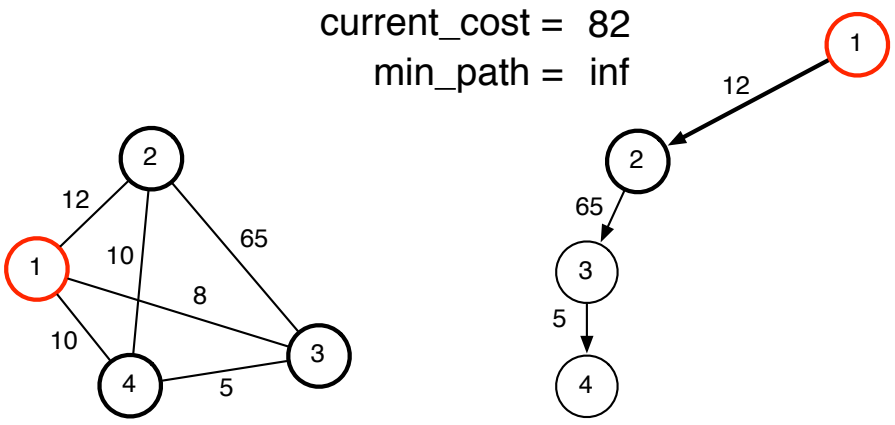
TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one



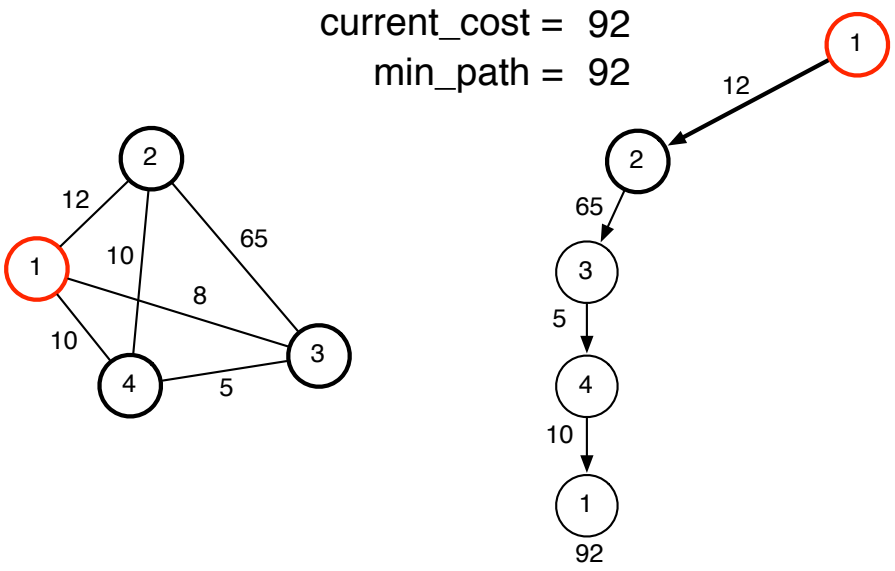
TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one



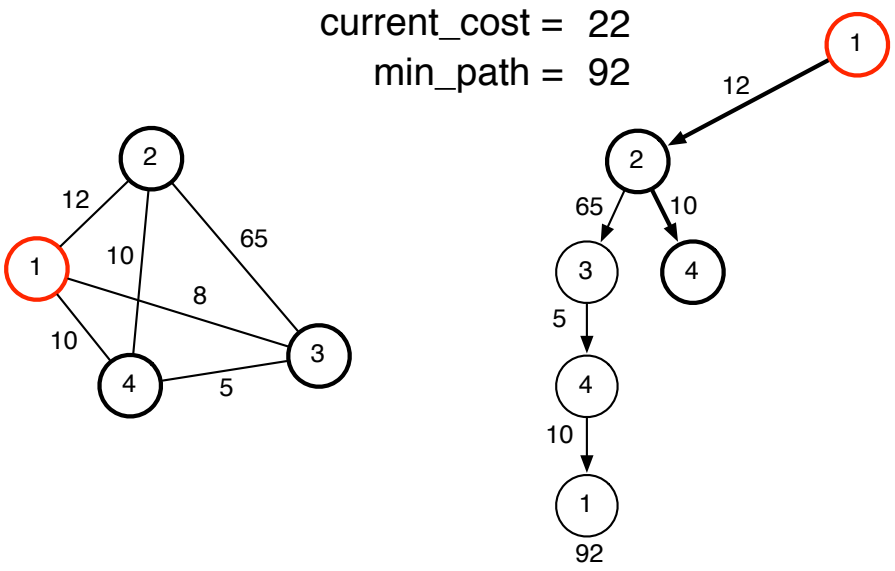
TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one



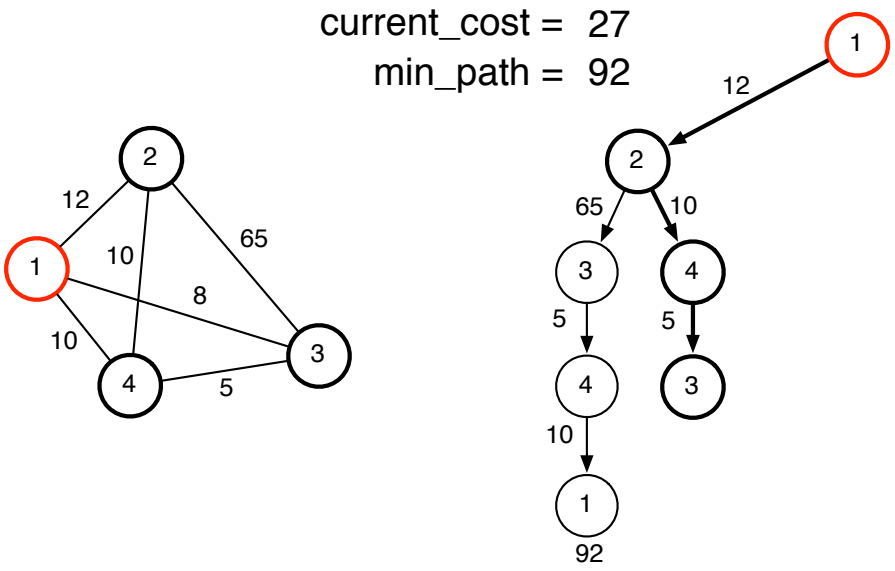
TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one



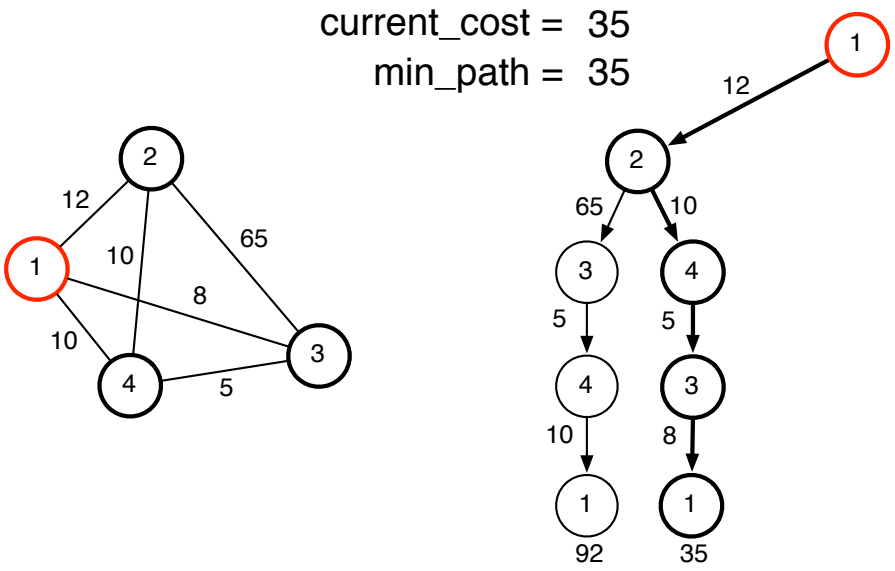
TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one



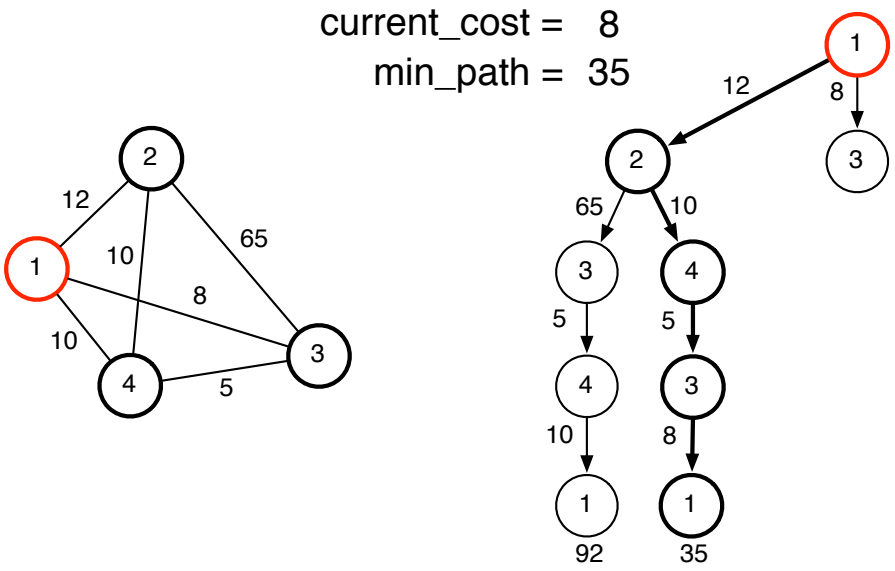
TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one



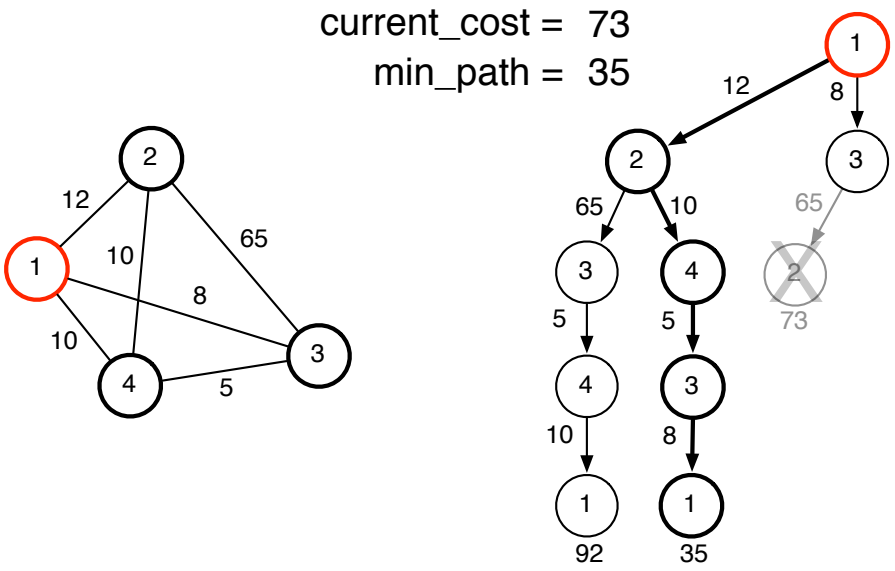
TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one



TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one

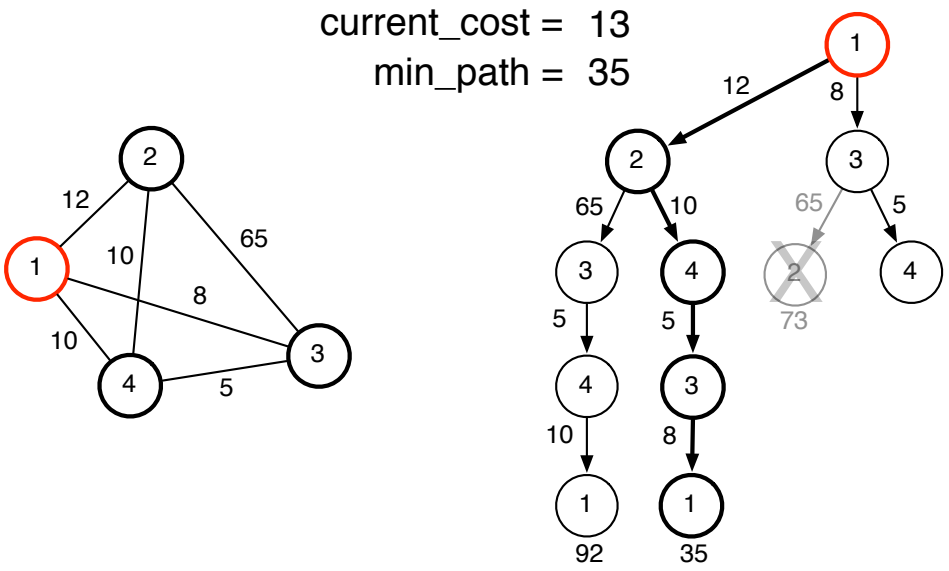


Irregular behavior

Pruning approach introduces irregularities into the search space!

TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one

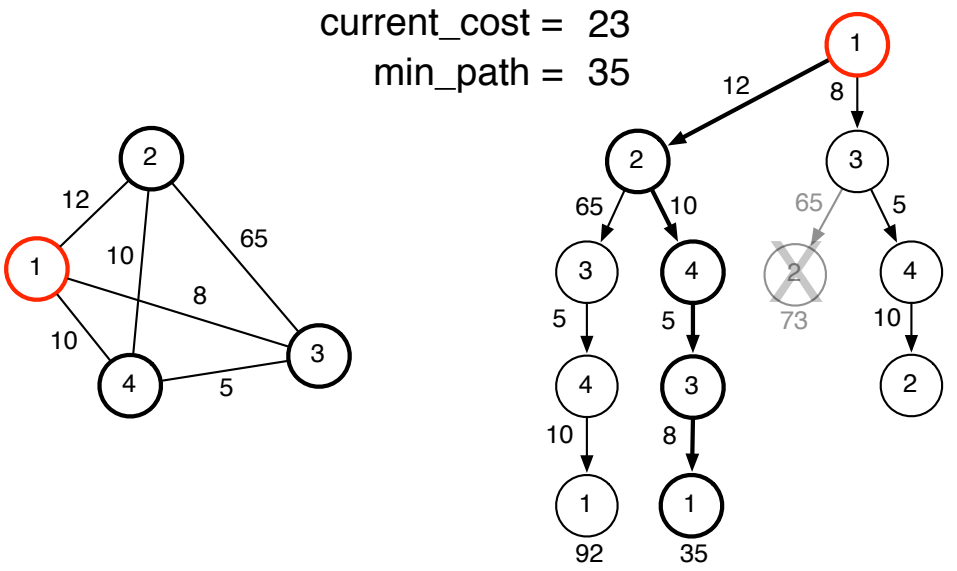


Irregular behavior

Pruning approach introduces irregularities into the search space!

TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one

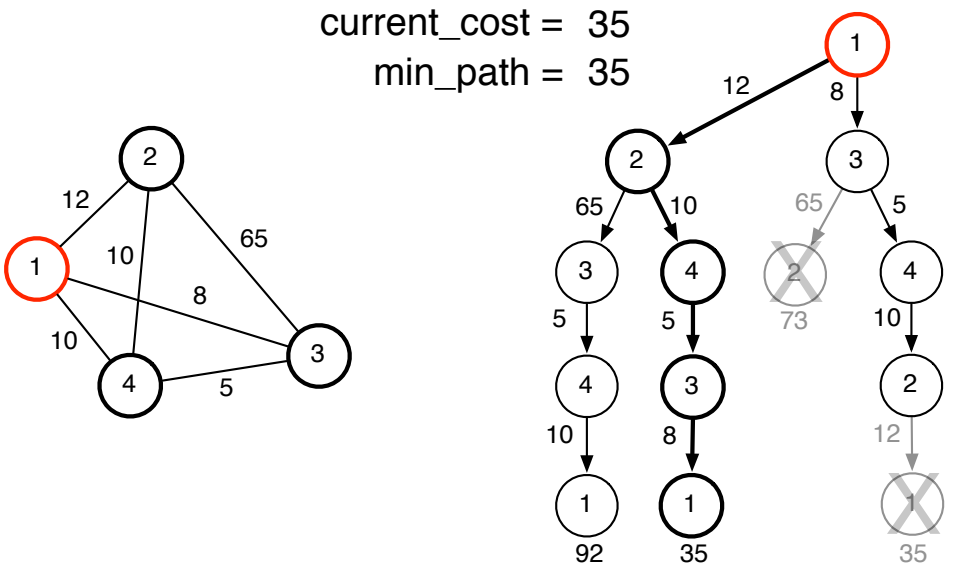


Irregular behavior

Pruning approach introduces irregularities into the search space!

TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one

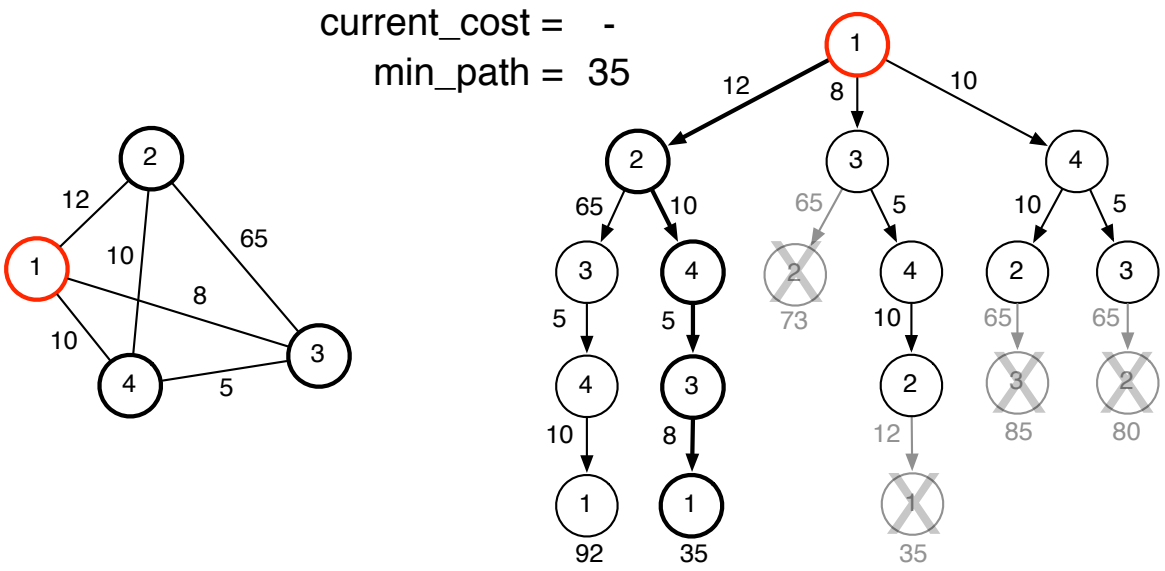


Irregular behavior

Pruning approach introduces irregularities into the search space!

TSP: Sequential algorithm

- Recursive method
- Depth-first traversal
- **Branch and bound**: doesn't explore paths that are longer than the current shortest one



Irregular behavior

Pruning approach introduces irregularities into the search space!

TSP: Multithreaded algorithm

- Generates tasks sequentially at the beggining
- Enqueues tasks in a centralized **queue of tasks**
- Threads atomically **dequeue tasks** and call *TSP_SOLVE*()

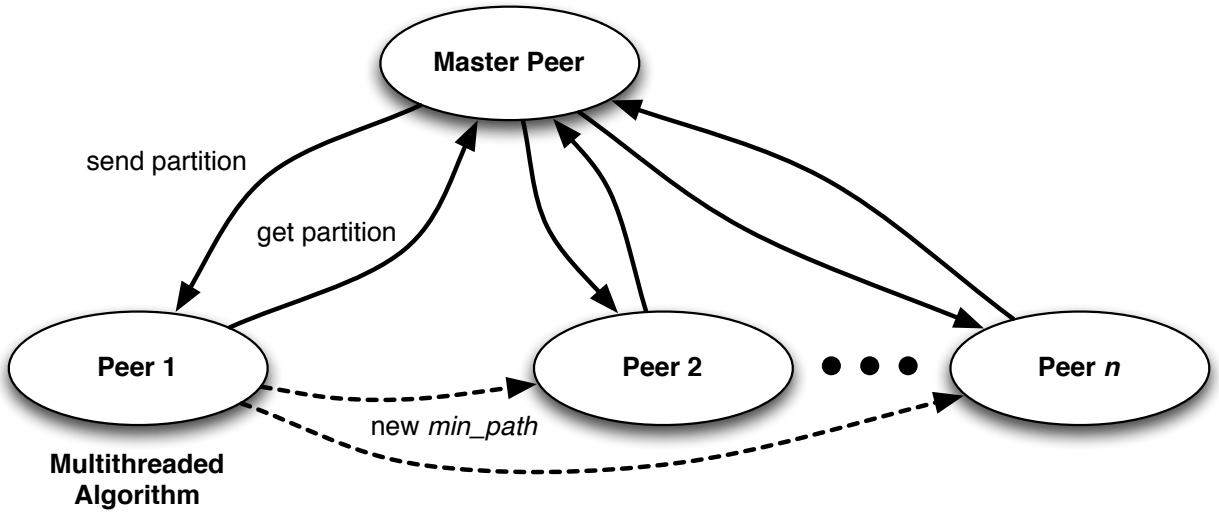
```
global queue, min_path
procedure GENERATE_TASKS(n_hops, last_city,
        current_cost, cities)
 if n_hops = max_hops
   then { task ← (last_city, current_cost, cities)
          ENQUEUE_TASK(queue, task)
   else { for each i ∈ cities
            do { if last_city = none
                   then last_cost ← 0
                   else last_cost ← costs[last_city, i]
                 new_cost ← curr_cost + last_cost
                 GENERATE_TASKS(n_hops + 1, i,
                        new_cost, cities\{i})

procedure DO_WORK()
 while queue ≠ ∅
   do { (last_city, current_cost, cities) ←
                 ATOMIC_DEQUEUE(queue)
        TSP_SOLVE(last_city, current_cost, cities)

main
 min_path ← ∞
 GENERATE_TASKS(0, none, 0, {1, 2, ..., n_cities})
 for i ← 1 to n_threads
   do SPAWN_THREAD(DO_WORK())
 WAIT_EVERY_CHILD_THREAD()
 output (min_path)
```

TSP: Distributed algorithm

- **Peers**: run the multithreaded algorithm
- **Master peer**: enqueues **partitions** in peers' local task queues
- **Partitions**: a set of tasks
- Peers **broadcast new shortest paths** when found
- The master peer **sends partitions of decreasing size at each request** to **decrease the imbalance between** the peers at the end of the execution



Adapting the TSP for manycores

Xeon E5 and Carma

- Multithreaded algorithm
- Shared variable *min_path* stores the shortest path and can be updated by all threads (locks needed)

Altix UV 2000

- Distributed algorithm
- **Broadcast** ➡ no explicit communication (locks and condition variables needed)
- **Thread and data affinity** to reduce NUMA penalties

Adapting the TSP for manycores

MPPA-256

- Distributed algorithm
- Communications between the master/peers ➡ **remote writes**
- **Absence of cache coherence**: worker threads inside peers might use a stale value of the *min_path* ➡ **performance loss**
- We used platform **specific instructions to bypass the cache** when reading/writing from/to *min_path*

Measurement methodology

Metrics

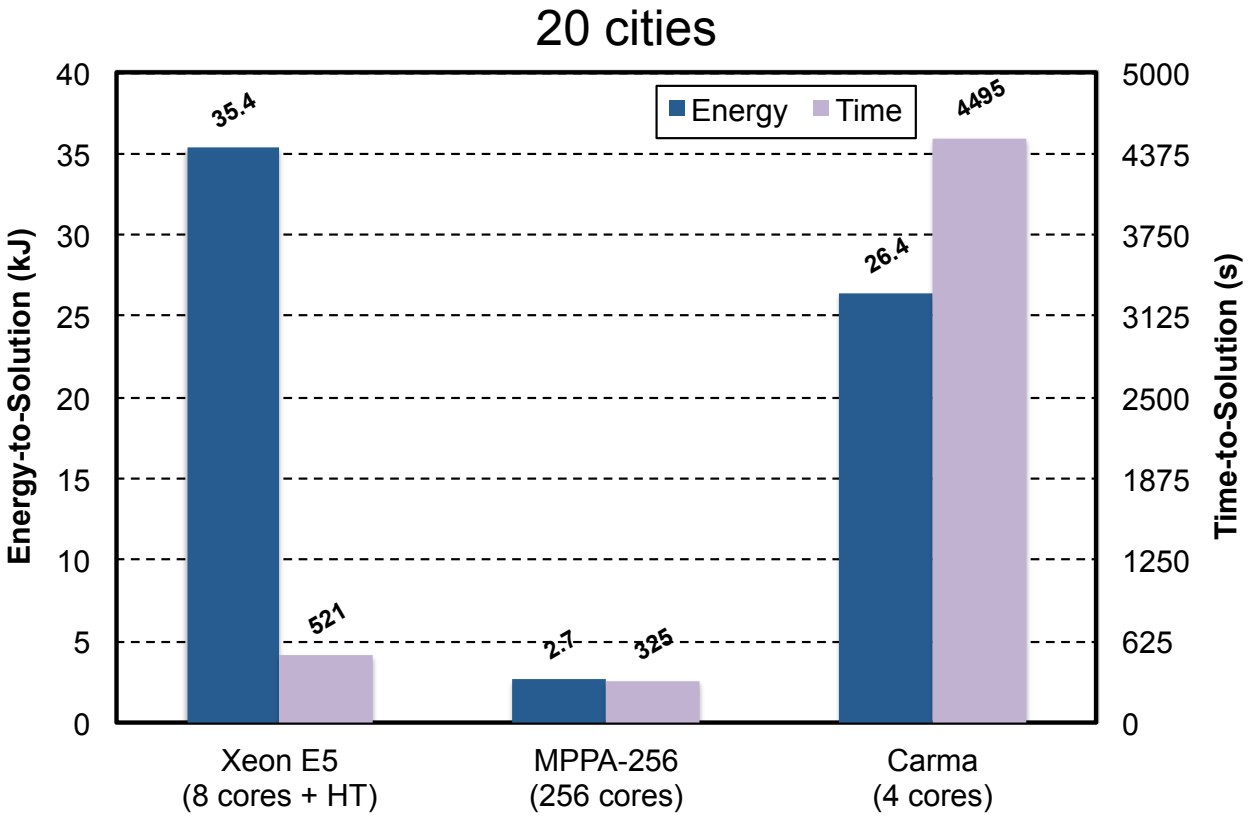
- **Time-to-solution**: time to reach a solution for a given problem
- **Energy-to-solution**: amount of energy to reach a solution for a given problem

Power and energy measurements

- **Xeon E5 and Altix UV 2000**: energy sensors (hw. counters)
- **MPPA-256**: energy sensors
- **Carma (Tegra 3)**: power consumption specification

	Xeon E5	Altix UV 2000	Carma	MPPA-256
Power (W)	68.6	1,418.4	5.88	8.26

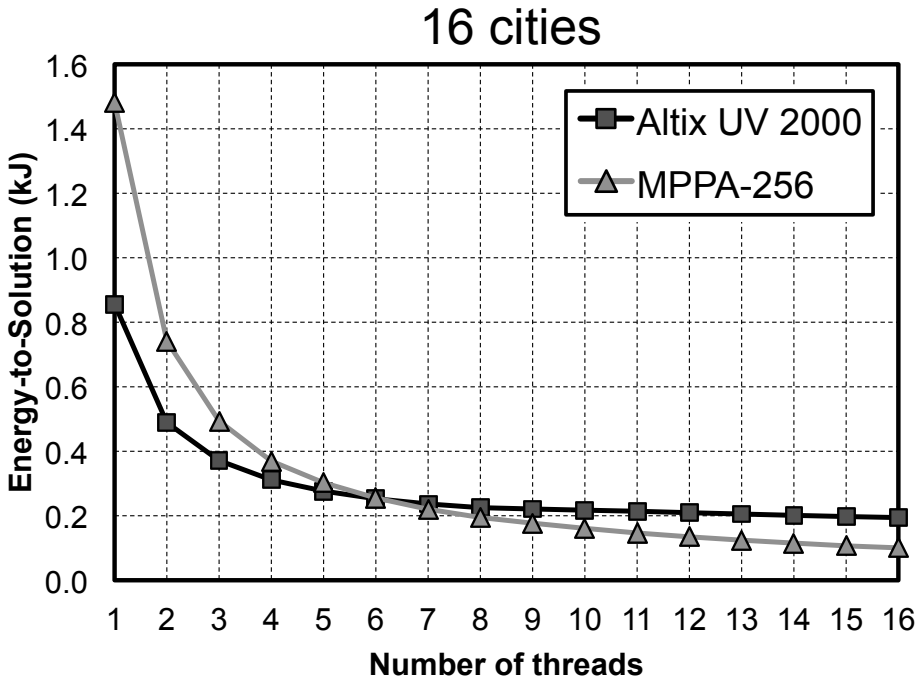
Chip-to-chip comparison: performance and energy



Results on MPPA-256:

- **Performance** ➡ ∼1.6x faster than Xeon E5
- **Energy** ➡ ∼10x less energy than Carma (Tegra 3)

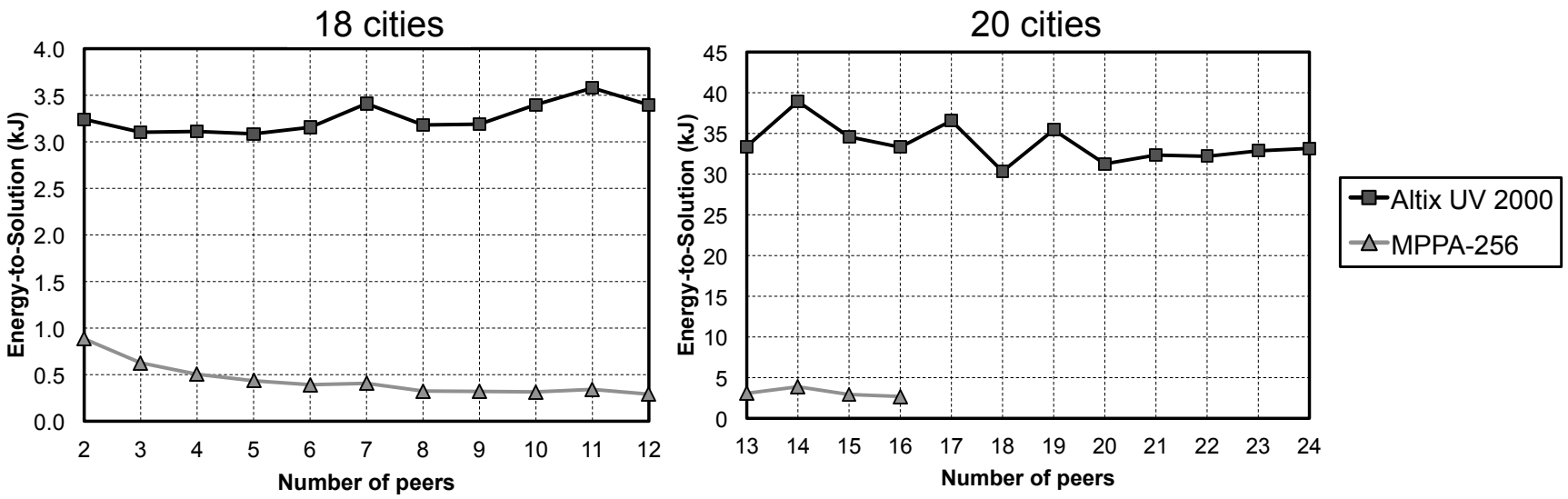
MPPA-256 single cluster vs. Altix UV 2000 single node



Single node/cluster power consumption

- **Altix**: from **27W** (1 thread) up to **68.6W** (16 threads)
- **MPPA-256**: from **3.7W** (1 thread) up to **4W** (16 threads)

Energy-to-solution: MPPA-256 vs. Altix UV 2000
Varying the number of clusters/nodes



- Peers: **NUMA nodes** (Altix UV 2000); **clusters** (MPPA-256)

MPPA-256 achieved much better energy-to-solution

- From 2 to 12 peers: 8.3x less energy
- From 13 to 16 peers: 11.3x less energy

Conclusions

For a fairly parallelizable application, such as the TSP

- MPPA-256 can be very competitive
- Better performance than Xeon E5 (∼1.6x)
- Better energy efficiency than Tegra 3 (∼9.8x)

However...

- It demands **non-trivial source code adaptations**, so that applications can efficiently use the whole chip
- **Absence of a coherent cache** considerably increases the implementation **complexity**

Future works

Assess how well the performance of this processor fares on applications with heavier communication patterns

- **Work in progress**: we are adapting a seismic wave propagation simulator developed by BRGM (France) to MPPA-256

Compare the computing and energy performance of MPPA-256 with other low-power processors

- **Mobile versions** of Intel's Sandy Bridge processors
- Investigate other manycores such as **Tilera's TILE-Gx**

Frameworks for heterogeneous platforms

- **Work in progress**: support for OpenCL on MPPA-256
- **JLPC**: Potential collaborations with Wen-Mei Hwu (UIUC), John Stratton (MCW-UIUC)

Questions?