Analysing the Impact of Parallel Programming Models in NoCs of Forthcoming CMP Architectures

Iván Pérez[1], Emilio Castillo[2,3], Ramón Beivide[1], Enrique Vallejo[1]

José Luis Bosque[1], Miquel Moretó[2,3], Marc Casas[2,3] and Mateo Valero[2,3]

[1] Universidad de Cantabria

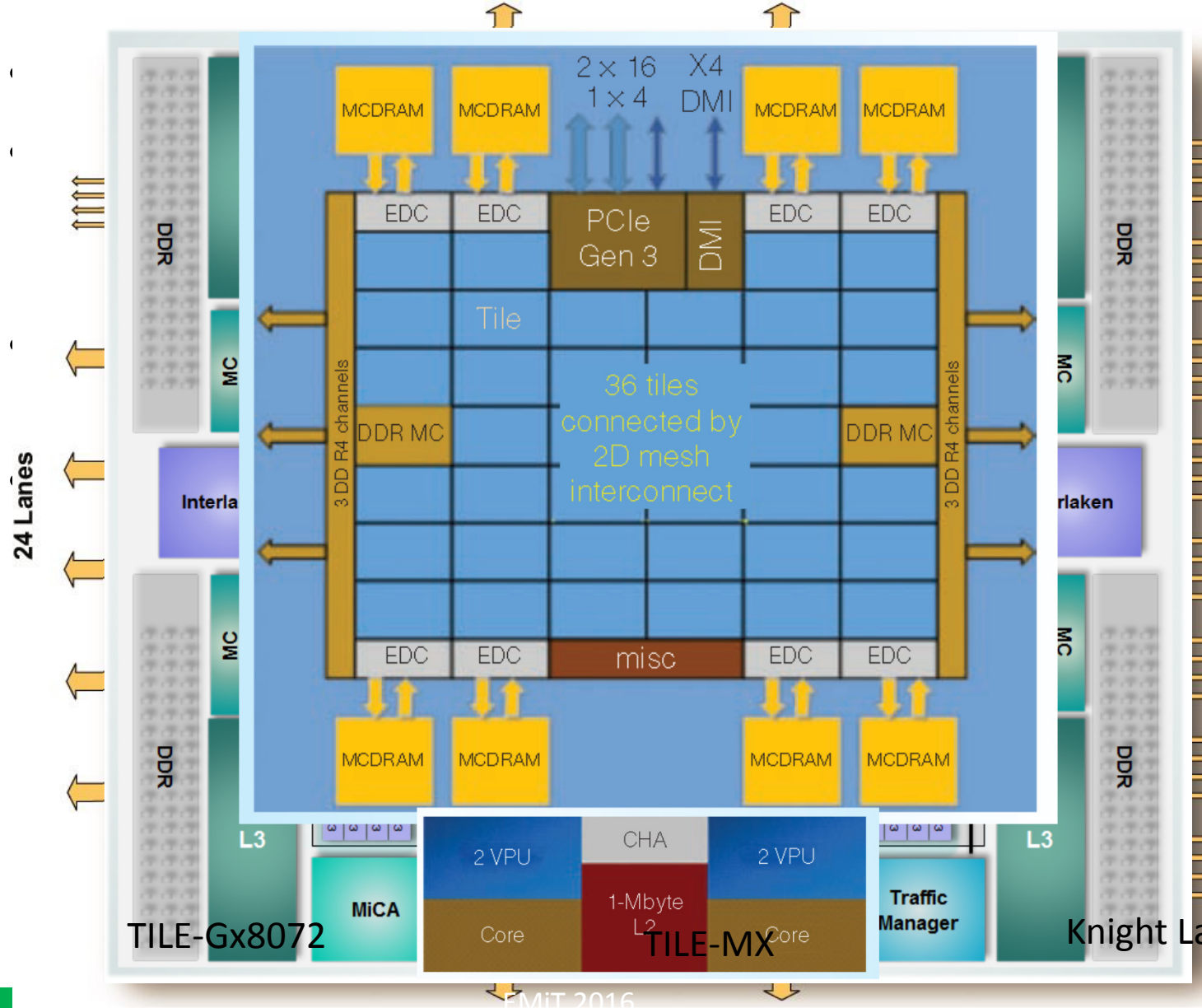
[2] Barcelona Supercomputing Center

[3] Universitat Politècnica de Catalunya

Outline

- Introduction
  - CMP Architectures
  - Parallel Programming Models
  - Objectives
- Methodology
- Results
- Conclusions

Introduction – Current Multiprocessors



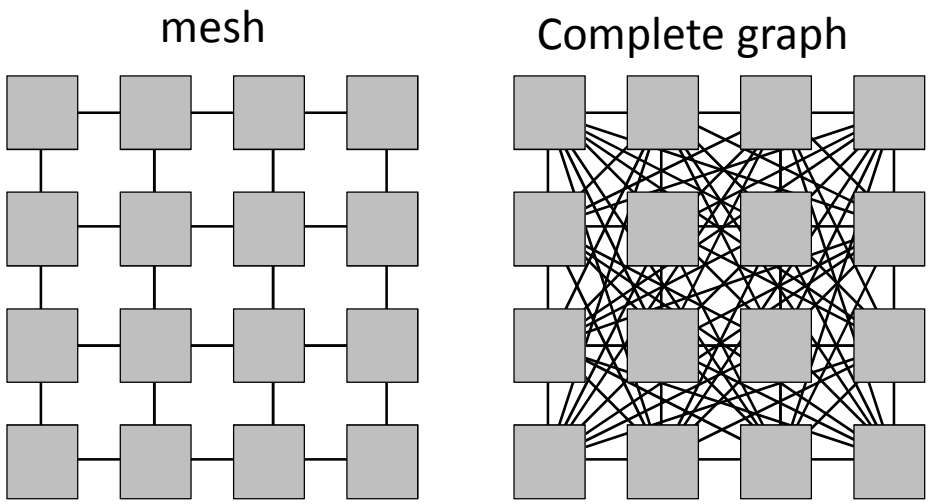
TILE-Gx8072 TILE-MX Knight Landing

Introduction - Parallel Programing Models

- A parallel programming model must provide efficiency in terms of **development** in addition to **good performance**.
- We compare two parallel programming models:
  - **Posix threads** (**pthreads**) which is based on threads. This model requires strict synchronization mechanism between threads to ensure correctness.
  - **OmpSs** that is a dataflow task programming model. Work units are tasks that are synchronized by data dependencies. It has a runtime (Nanos++) that tracks dependencies and schedules task executions.

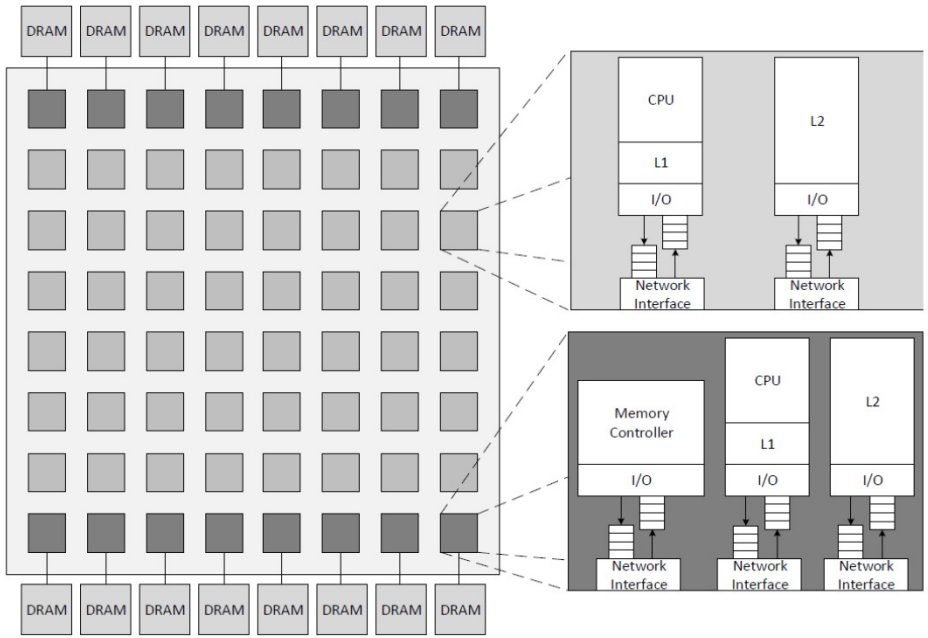
Introduction – Objectives

- Evaluate both programming models in next-generation CMP architectures.
- Analyse the impact of both programming models in the NoC utilization. Two NoC topologies are considered:
  - Traditional 2D mesh: Typical NoC topology, widely used.
  - Complete graph: Best-case, unrealistic implementation.
    - Only 1 hop between routers.
    - Requires huge routers (large port count).



Methodology - Environment and Configuration

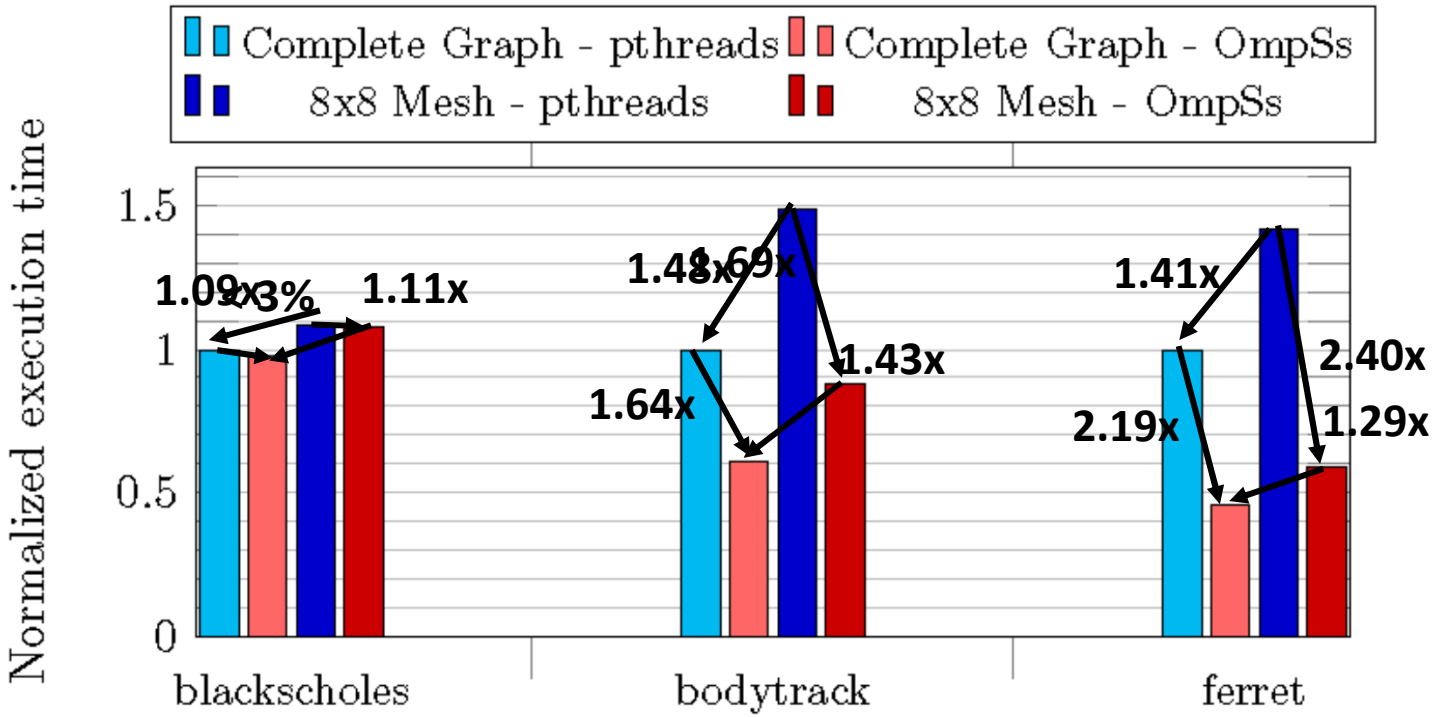
- We use gem5 to simulate 64 x86 core systems, with two levels of cache.
- The L2 cache is a 64 bank shared NUCA.
- Three Parsec benchmarks: Blackscholes, Bodytrack and Ferret



Parameter	Value
CPU units	64
CPU ISA	x86
CPU model	O3
CPU frequency	100 MHz - 2 GHz
Ruby frequency	1 GHz
Coherence protocol	MESI
Memory controllers	16
Network model	5-stage Garnet router
Topology	8x8 mesh and 64 complete graph
Virtual network (VN)	3
Virtual channels per VN	1
Buffers per port	10 flits
Flit size	16 B
Block size	64 B
Message control size	8 B
L1I Size	32 KB
L1D Size	64 KB
L1 Latency	1 Ruby cycle
L2 Size	64 banks of 512 KB
L2 Latency	15 Ruby cycles
DRAM type	DDR3-1600

Results - Execution time

- Programming models:
  - Negligible difference in Blackscholes.
  - Huge speedup in the applications that use thread pools.
- Network topologies:
  - Huge impact in the system performance.
  - In some cases pthreads has higher sensitivity to the network performance.



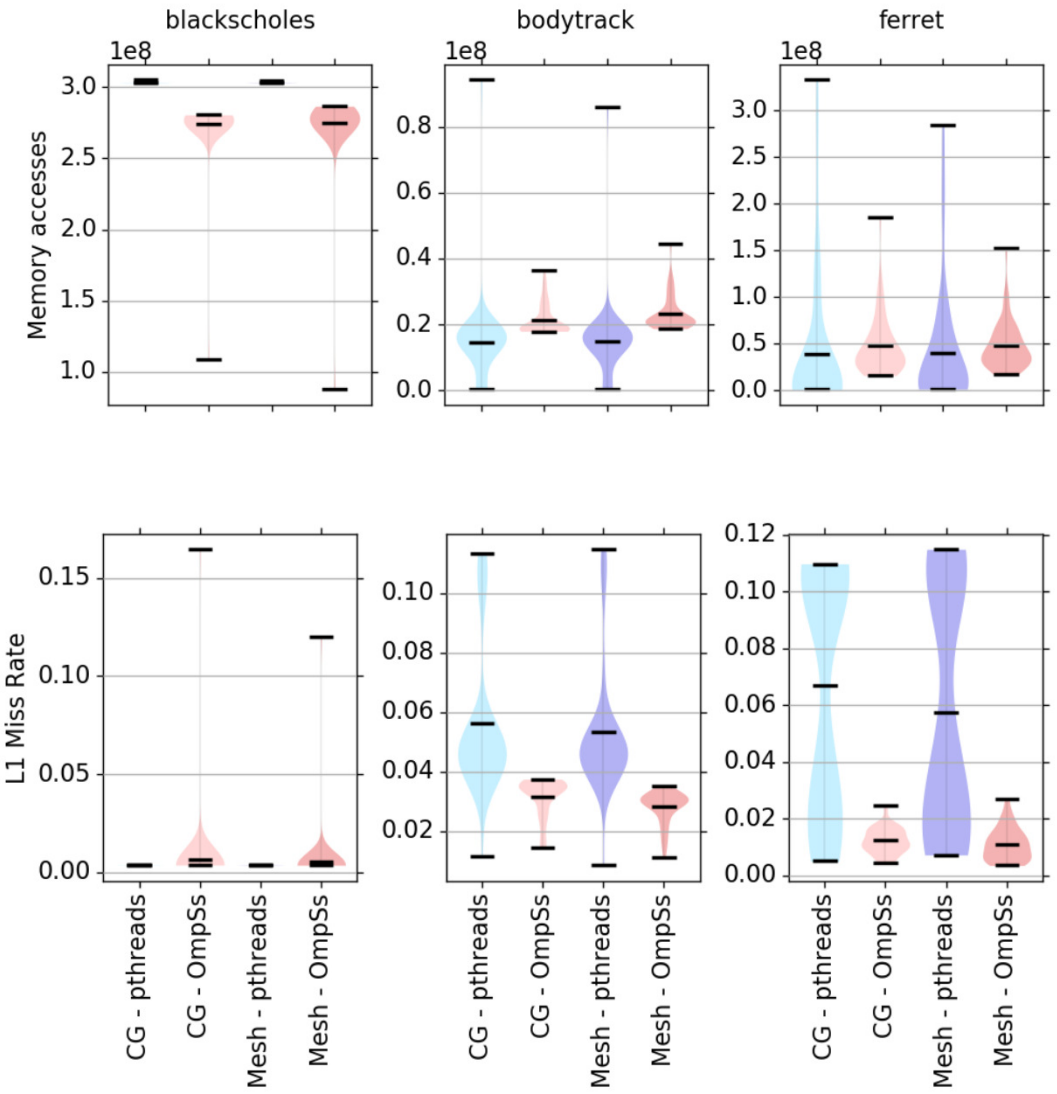
Relating Programming Models with NoC Utilization

- The programming model influences on:
  - Number of executed load/stores
  - Miss rates: data locality
- The number of executed load/stores and miss rates determine the L1 and L2 misses which, in turn, determine the network load (injected flits).
- Network performance depends on:
  - Injection rate: injected flits per core per unit time
  - Latencies:
    - Injection latency
    - Network latencies (average network distance)

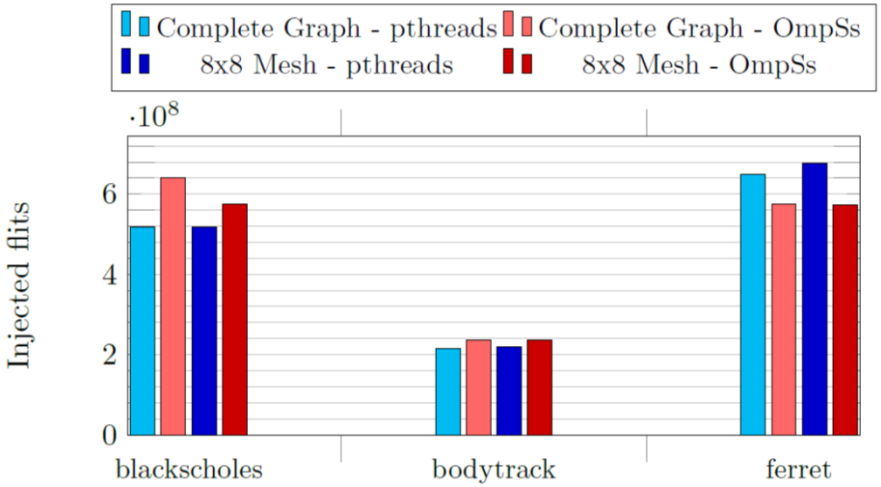
Results - Memory accesses and miss rates

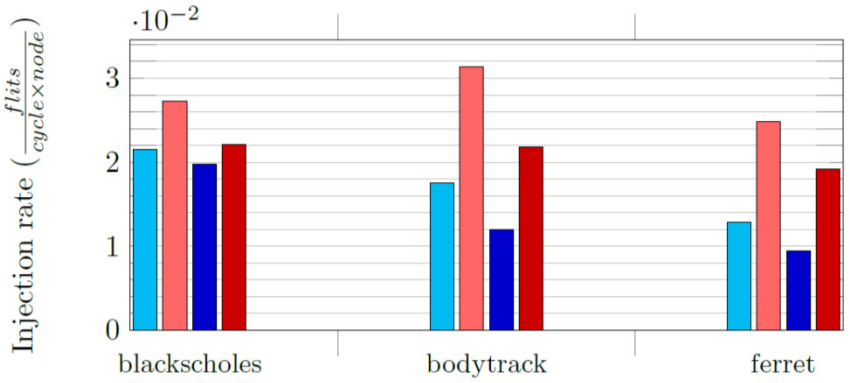
- Distribution of Loads/Stores (64 cores):
  - 10% lower in Blackscholes
  - 45% and 21% bigger in Bodytrack and Ferret.
  - Less variability (except in Blackscholes) so the load is better balanced

- L1 data miss rate:
  - 3% and 7% lower miss rates for Bodytrack and Ferret respectively.
  - Better exploit of locality.

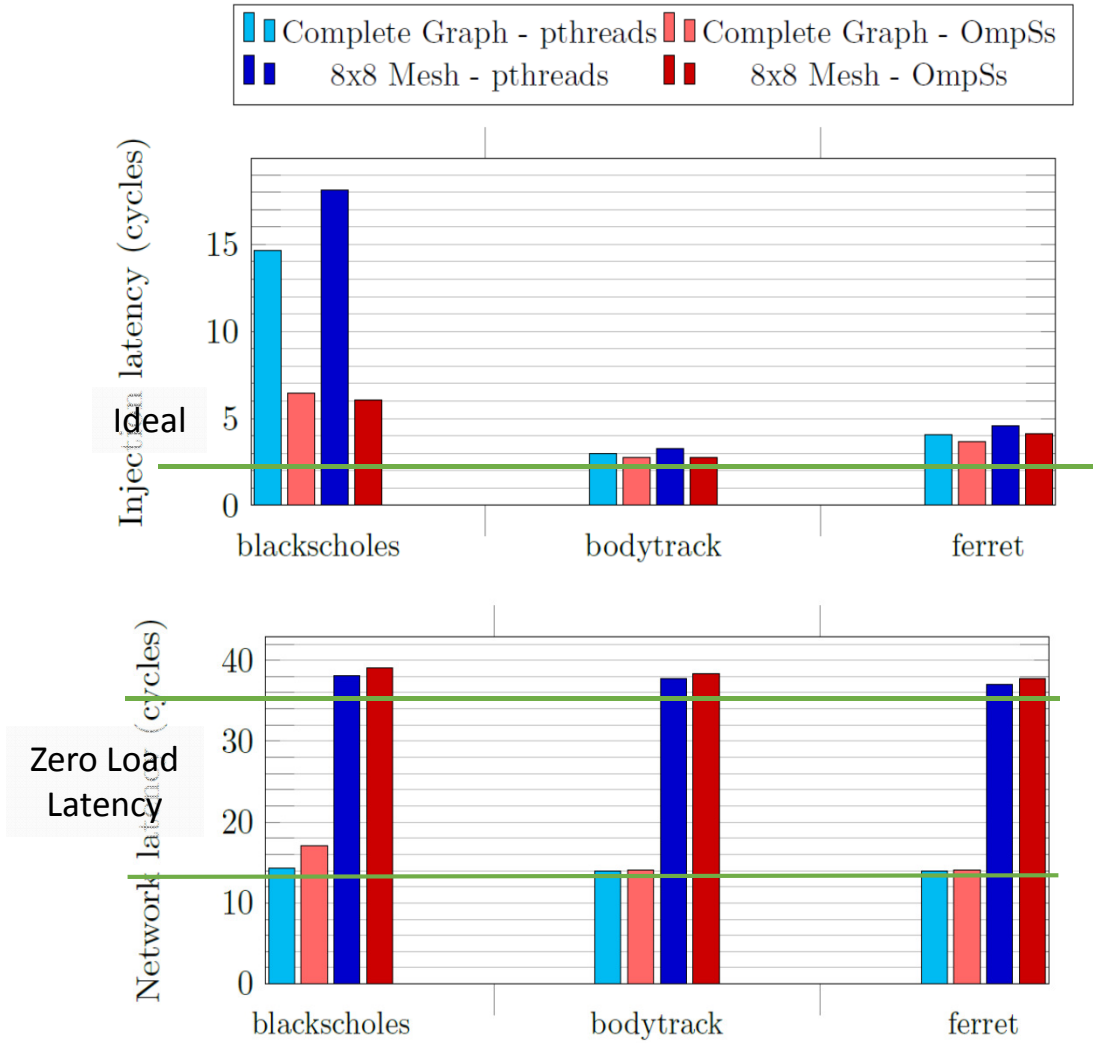


Results – Injected Flits and Injection Rate





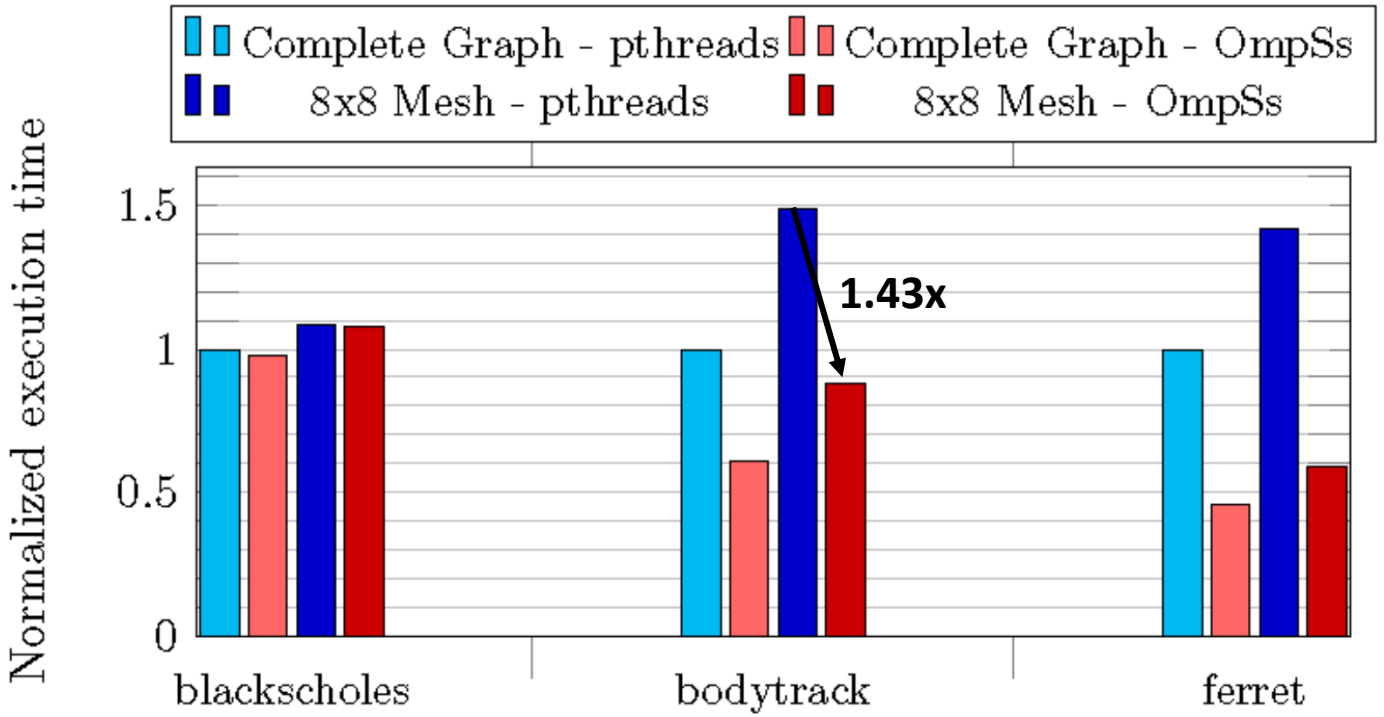
Results - NoC latencies



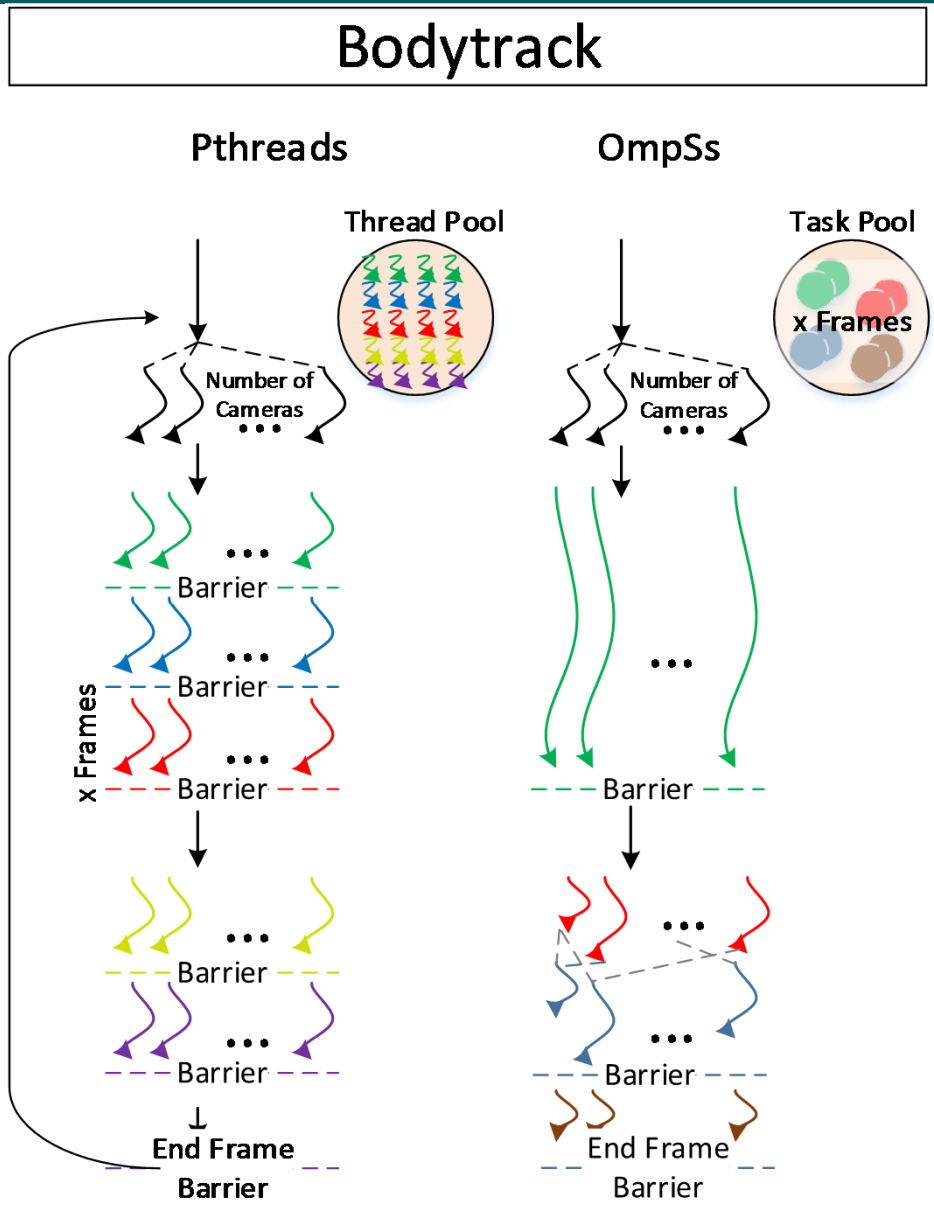
Conclusions and Future Work

- Conclusions:
  - OmpSs clearly outperforms pthreads.
  - The impact of the NoC in the system performance is significant: between 1.09 and 1.48 of speedup can be achieved in the experiments comparing complete graph vs mesh.
  - The most important network parameter is the average distance which will determine the zero load latency.
  - OmpSs stresses more the network (higher injection rates). This can lead in a higher sensitivity to the NoC design.
- Ongoing work:
  - Detailed evolution of statistics along time in order to characterize NoC traffic.
  - Evaluation on concentrated meshes.

Results - Execution time

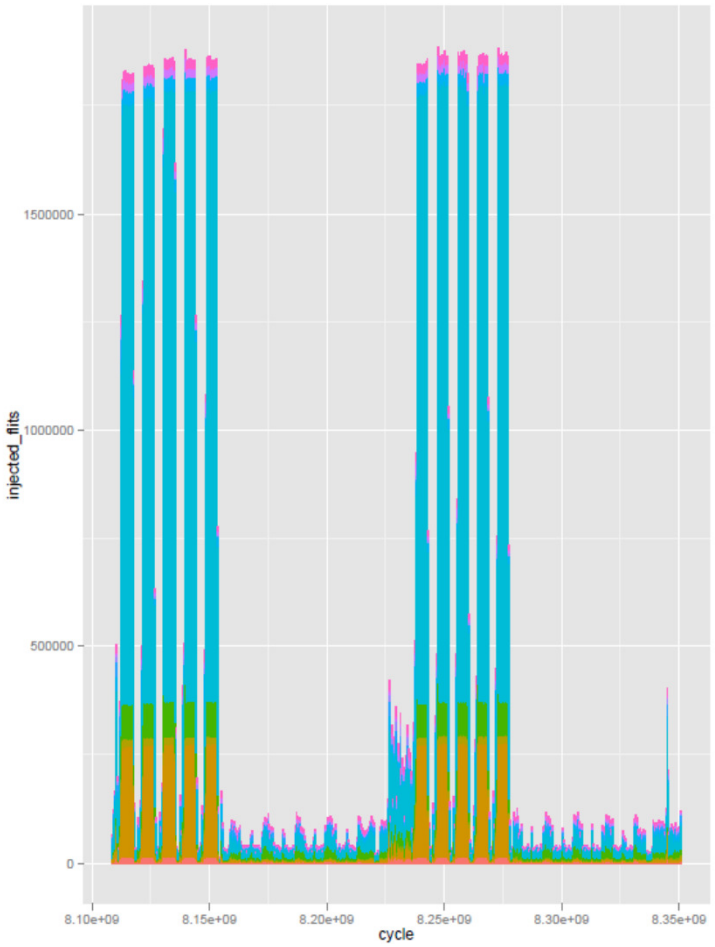


Bodytrack

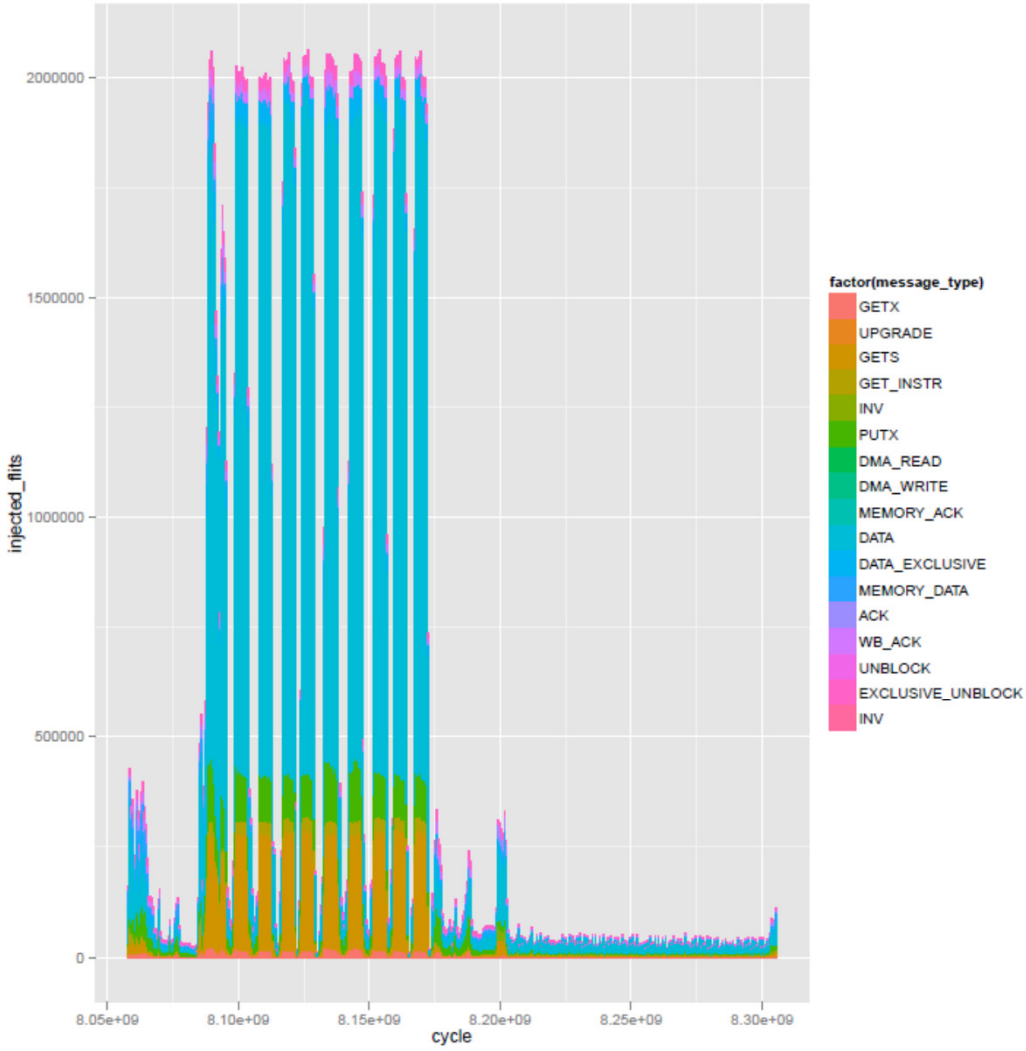

Bodytrack
Pthreads
OmpSs
Thread Pool
Task Pool
x Frames
Number of Cameras
Number of Cameras
Barrier
Barrier
Barrier
Barrier
Barrier
Barrier
Barrier
x Frames
End Frame Barrier
End Frame Barrier

Injected flits along time

Pthreads



OmpSs



Analysing the Impact of Parallel Programming Models in NoCs of Forthcoming CMP Architectures

Iván Pérez[1], Emilio Castillo[2,3], Ramón Beivide[1], Enrique Vallejo[1]

José Luis Bosque[1], Miquel Moretó[2,3], Marc Casas[2,3] and Mateo Valero[2,3]

[1] Universidad de Cantabria

[2] Barcelona Supercomputing Center

[3] Universitat Politècnica de Catalunya

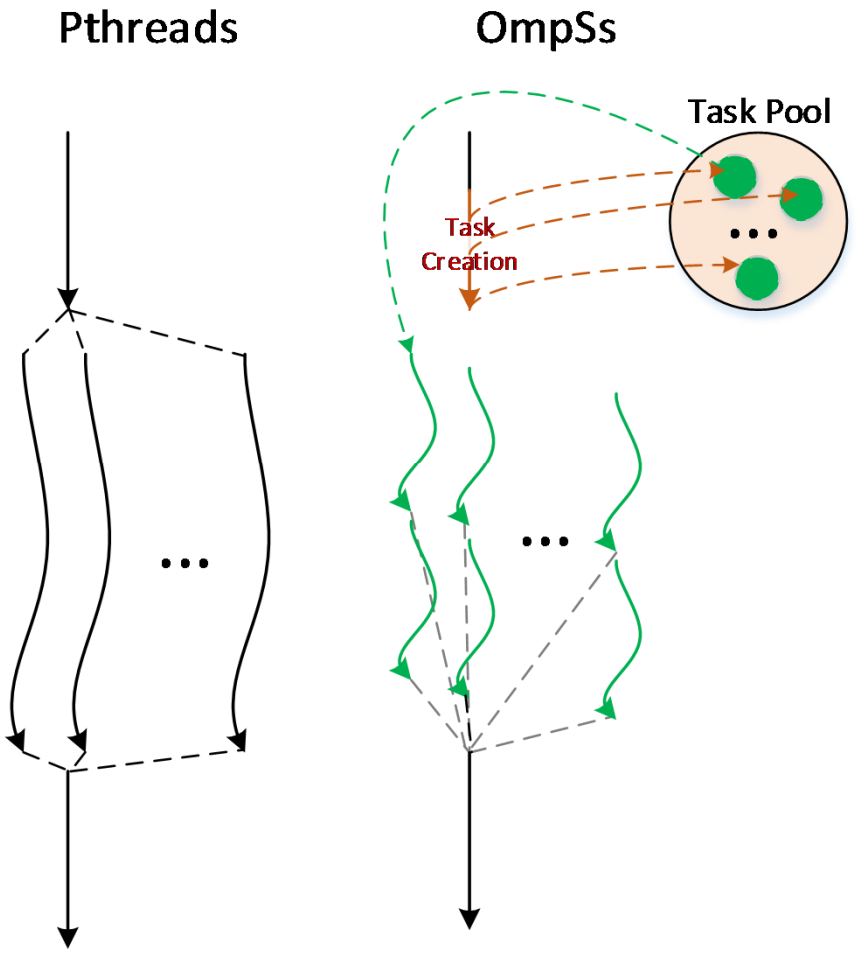
Methodology - Workloads

- Three PARSEC benchmarks are executed via simulations:
  - Blackscholes: embarrassing parallel application.
  - Bodytrack: parallelization on 5 kernels synchronized by barriers.
  - Ferret: parallelization based on a 6-stage pipeline.

Benchmark	Input set
Blackscholes	1,048,576 options
Bodytrack	2 frame, 2,000 particles
Ferret	64 queries, 13,787

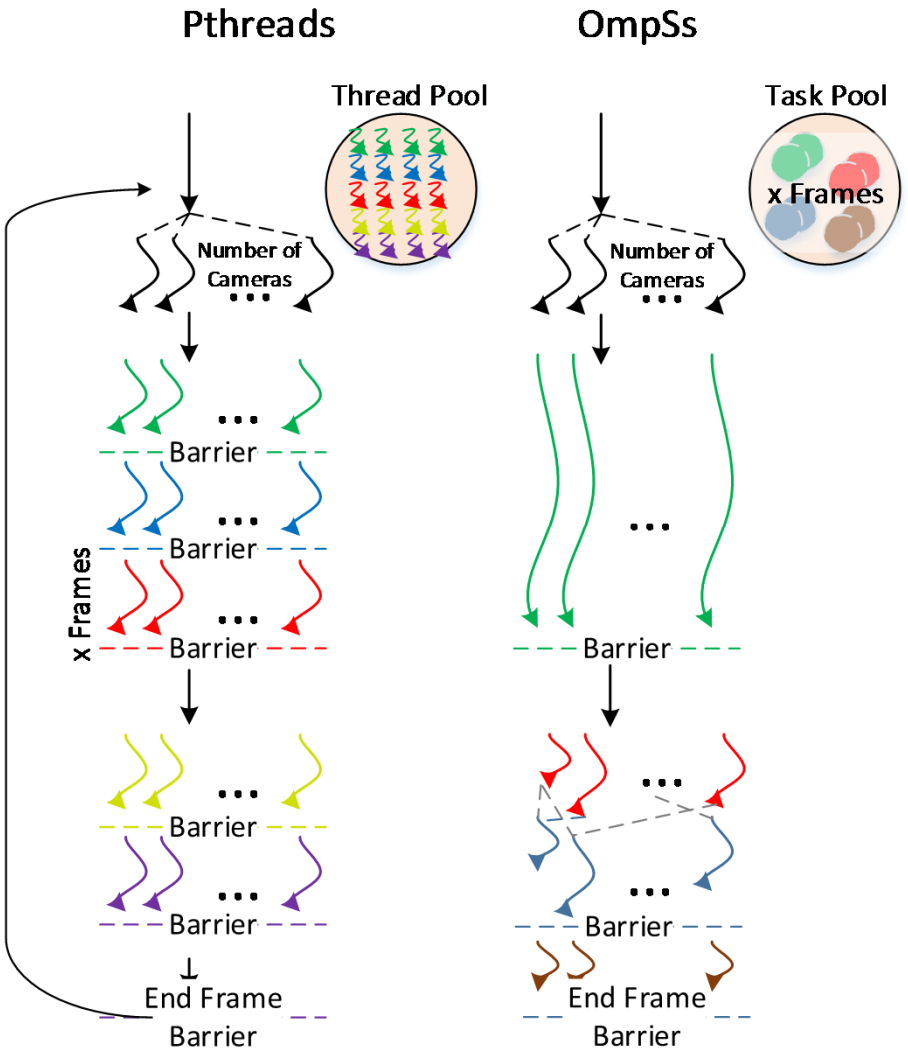
Workloads - Blackscholes

- Embarrassing parallel application.
- Pthreads divides the load by the number of the available threads.
- OmpSs increases the granularity in order to improve load balance.



Workloads - Bodytrack

- There are 5 parallel kernels in pthreads. The implementation uses thread pools and barriers at the end of each kernel to control the data flow. Each frame is processed secuencially.
- OmpSs reduces the granularity fusing the 3 kernels of the first stage. In addition the tasks of all frames are created as soon as possible adding the possibility of execute concurrently tasks of different frames.



Workloads - Ferret

- Pthreads implements a pipeline of 6-stages using thread pools.
- OmpSs implementation is very similar to the pthreads one. As soon as a queried image is found all the tasks of the pipeline for that query are created.