Auto-tuned nested parallelism: a way to reduce the execution time of scientific software in NUMA systems

Jesús Cámara, **Javier Cuenca**,
Luis P. García, Domingo Giménez

Scientific&Parallel Computing Group
University of Murcia, SPAIN

parallelum

Introduction

- Scientific and engineering problems are solved with large parallel systems
- In some cases those systems are NUMA
  - A large number of cores
  - Share a hierarchically organized memory
- Kernel of the computation for those problems: BLAS o similar
  - Efficient use of kernels → a faster solution of a large range of scientific problems
- Normally: multithreaded BLAS library optimized for the system is used, but:
  - If the number of cores increases → the degradation in the performance grows
- In this work:
  - Analysis of the behaviour in NUMA of an example of high-level routine: a LU factorisation
  - An improved scheme: [multithreaded `dgemm` of BLAS + OpenMP] → nested parallelism
  - An auto-tuning method → a reduction in the execution time

Outline

- Introduction
- Computational systems
- The software
- Motivation
- Automatic optimisation method
  - Design phase
  - Installation phase
  - Execution phase
- Conclusions and future work lines

Computational systems

- **Pirineus**
  - SGI Altix UV 1000
  - 224 nodes Intel Xeon six-core serie 7500
  - Total: 1344 computing cores
- **Ben**
  - HP Integrity Superdome with architecture NUMA
  - 64 nodes Itanium-2 with 4 CPUs dual core
  - Total: 128 computing cores
- **Saturno**
  - A server SYS-8026B-TRF 2U supermicro
  - 4 nodes Intel six-core NEHALEM-EX 6C E7530
  - Total: 24 computing cores

Outline

- Introduction
- Computational systems
- The software
- Motivation
- Automatic optimisation method
  - Design phase
  - Installation phase
  - Execution phase
- Conclusions and future work lines

The software

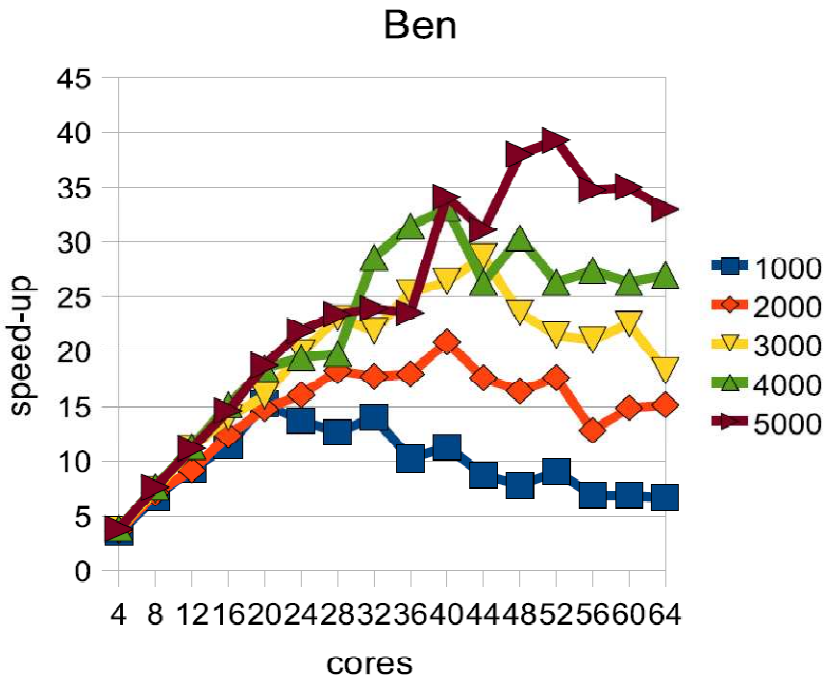
- **Intel MKL toolkit 10.2**
  - Multithreaded
  - Dynamic parallelism enabled → number of threads decided by the system
  - Dynamic parallelism disabled → number of threads decided by the user
- **C compiler**: Intel `icc` version 11.1 in Ben and Pirineus, 12.0 in Saturno
- **Kernel Routine**: MKL BLAS double precision matrix multiplication: `dgemm`
- **High level routine**: A block LU factorisation. Two different implementation schemes:
  - The traditional → with [multithreaded `dgemm`]
  - The improved → with nested parallelism [OpenMP+multithreaded `dgemm`]
- **Matrices multiplication** (*AB* = *C*) with nested parallelism (*q* x *p* threads):
  - *q* threads OpenMP. Each OpenMP thread:
    - multiplies a block of rows of A by the whole B → a block of rows of C
    - uses *p* MKL threads inside

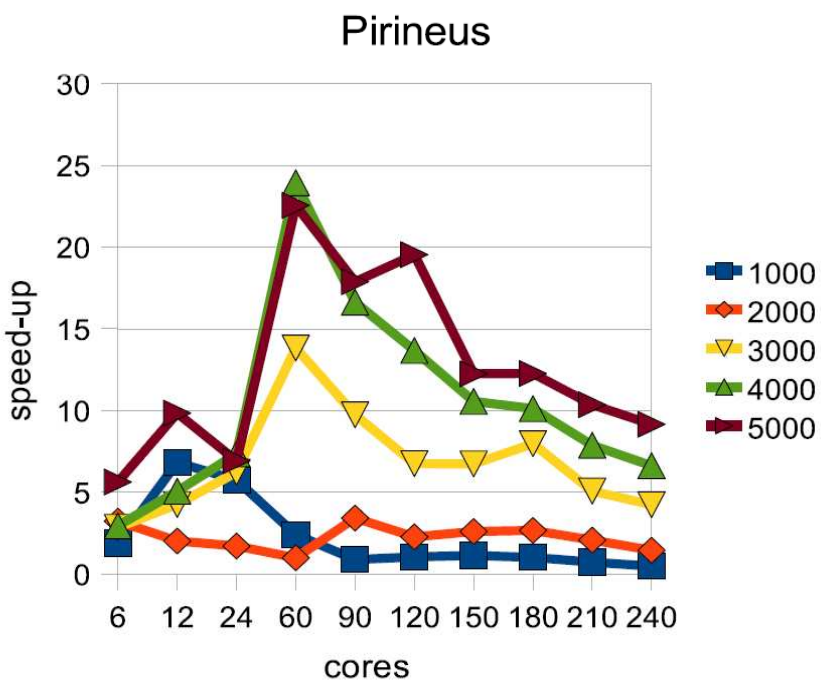
Outline

- Introduction
- Computational systems
- The software
- Motivation
- Automatic optimisation method
  - Design phase
  - Installation phase
  - Execution phase
- Conclusions and future work lines

Motivation

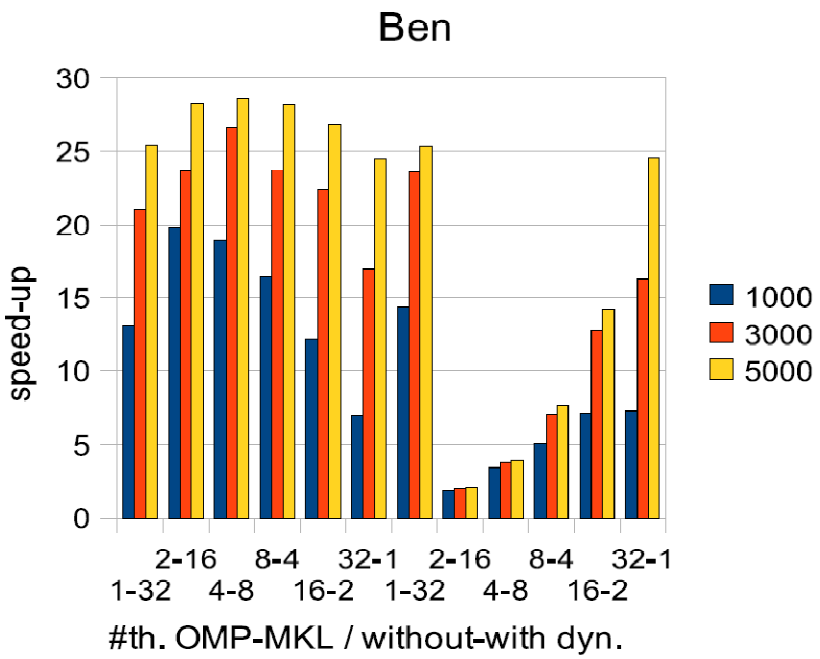
- Using a multithreaded version of BLAS → the `dgemm` MKL routine
- Optimum numbers of threads changes:
  - from one platform to another
  - for different problem sizes.
- Default option (number of threads = available cores) is not good

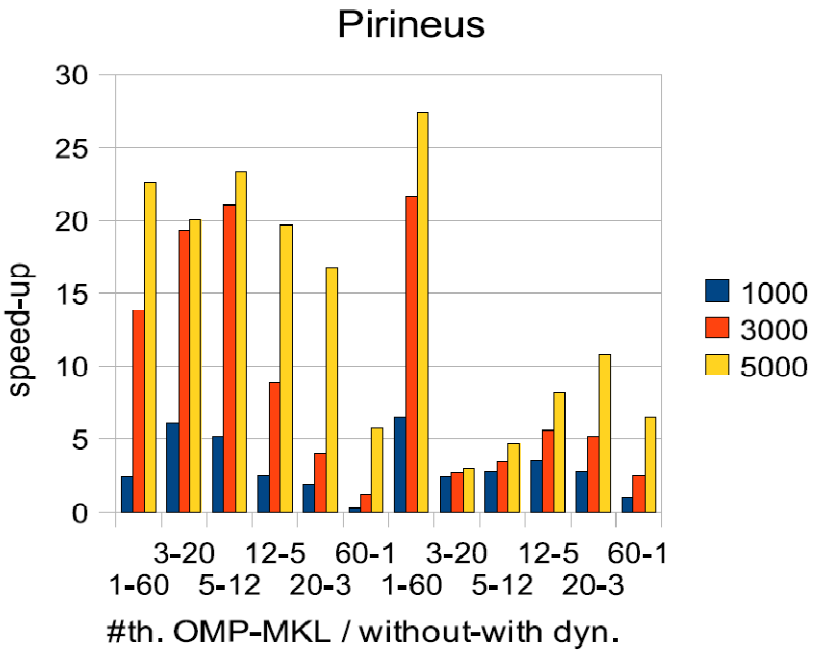




Motivation

- **Dynamic Selection of threads:**
  - Improvement in the speed-up increases with the number of OpenMP threads
  - Number of MKL threads used is just one
- **No Dynamic Selection of threads:**
  - Bigger speed-ups are obtained
  - Number of OpenMP threads grows → an increase of the speed-up until a maximum
  - So, a large number of cores → a good option to use a high number of OpenMP threads





Outline

- Introduction
- Computational systems
- The software
- Motivation
- Automatic optimisation method
  - Design phase
  - Installation phase
  - Execution phase
- Conclusions and future work lines

Automatic optimisation method

- **A**utomatic **T**uning **S**ystem (**ATS**) focused on modelling the execution time

$$T_{exe} = f(n, SP, AP)$$

  - ***n***: the **problem size**
  - ***SP***: **System Parameters**. Characteristics of the platform (hardware + basic installed libraries)
  - ***AP***: **Algorithmic Parameters**. Values chosen by the **ATS** to reduce the execution time

- <u>An adaptation to large NUMA platforms:</u>
  - An arithmetic operation: data access time depends on the **relative position in memory space**
    - Data can be in the closest memory of the processor or in that of another processor
    - The interconnection network could be non homogeneous
  - Therefore
    - → those data could be at **different distances** from the processor that needs them
- → the access time is modelled with a **hierarchical vision of the memory**

  - It is also necessary to take into account the **migration system** of the platform

Outline

- Introduction
- Computational systems
- The software
- Motivation
- Automatic optimisation method
  - Design phase
  - Installation phase
  - Execution phase
- Conclusions and future work lines

Automatic optimisation method

Design phase: modelling the execution time of the routine

Modelling 1-Level: MKL multithreading dgemm without generating OpenMP threads

- Model:

$$T_{dgemm} = \frac{2n^3}{p} k_{dgemm}$$

- ***AP***: ***p*** → number of threads inside dgemm
- ***SP***: k_{dgemm} → time to perform a basic operation inside dgemm (load&store included)
  - Taking into account NUMA and its data migration system:

$$k_{dgemm} = \alpha k_{dgemm\_NUMA}(p) + (1-\alpha) k_{dgemm\_M1}$$

  - k_{dgemm_M1} → operation time when data are in the clostest memory to the core (mem. level 1)
  - k_{dgemm_NUMA} → operation time when data are in any level of the RAM memory
  - α → weighting factor
    - directly proportional to the use by each of the ***p*** thread of data assigned to the other ***p-1*** threads
    - inversely proportional to the reuse degree of data carried out by this routine (dgemm → n^3/n^2)

$$\alpha = min\left\{1, \frac{p(p-1)}{n^3/n^2}\right\}$$

Automatic optimisation method

Design phase: modelling the execution time of the routine

Modelling 1-Level: MKL multithreading dgemm without generating OpenMP threads

- Platform:
  - H → number of memory levels
  - c_l → number of computing cores with similar access speed to the level l, with $1 \leq l \leq H$
- k_{dgemm_NUMA} value can be modelled, depending on p:
  - If $0 < p \leq c_1$:

$$k_{dgemm\_NUMA}(p) = k_{dgemm\_M1}$$

  - else if $c_1 < p \leq c_2$:

$$k_{dgemm\_NUMA}(p) = \frac{c_1 k_{dgemm\_M1} + (p - c_1) k_{dgemm\_M2}}{p}$$

  - …, in general, if $c_{H-1} < p \leq c_H$:

$$k_{dgemm\_NUMA}(p) = \frac{\sum_{l=0}^{H-2} (c_l - c_{l-1}) k_{dgemm\_Ml} + (p - c_{H-1}) k_{dgemm\_MH}}{p}$$

Automatic optimisation method

Design phase: modelling the execution time of the routine

Modelling 2-Level: OpenMP threads + MKL multithreading dgemm

- Model:

$$T_{2L\_dgemm} = \frac{2\frac{n}{q}nn}{p} k_{2L\_dgemm} = \frac{2n^3}{R} k_{2L\_dgemm}$$

- ***AP*** →***R = p*** x ***q*** **threads** interactuating
  - ***p*** → Number of threads inside the MKL routine dgemm
  - ***q*** → Number of OpenMP threads

- ***SP*** → k_{2L_dgemm} : time to carry out a basic operation

$$k_{2L\_dgemm} = \alpha k_{2L\_dgemm\_NUMA}(R,p) + (1-\alpha) k_{2L\_dgemm\_M1}$$

$$k_{2L\_dgemm\_NUMA}(R,p) = \frac{k_{dgemm\_NUMA}(R) + k_{dgemm\_NUMA}(p)}{2}$$

$$\alpha = min\left\{1, \frac{R(R-1)}{n^3/n^2}\right\}$$

Outline

- Introduction
- Computational systems
- The software
- Motivation
- Automatic optimisation method
  - Design phase
  - Installation phase
  - Execution phase
- Conclusions and future work lines

Automatic optimisation method

Installation phase: experimental estimation of the *SP* values

- General process: calculating the ***SP*** values that appear in the model
- ***SP*** values to calculate: k_{dgemm_M1} ,..., k_{dgemm_MH}
- For each memory level ***l***, from ***l*=1** until ***H***:

1. Executing `dgemm` → experimental execution time (experimental T_{dgemm}):
   - for a fixed problem size, ***n***
   - for a number of threads, p_l, with $c_{l-1} < p_l \leq c_l$

Automatic optimisation method

Installation phase: experimental estimation of the *SP* values

- General process: calculating the ***SP*** values that appear in the model
- ***SP*** values to calculate: k_{dgemm_M1} ,..., k_{dgemm_MH}
- For each memory level ***l***, from ***l*=1** until ***H***:

1. Executing `dgemm` → experimental execution time (experimental T_{dgemm}):
   - for a fixed problem size, ***n***
   - for a number of threads, p_l, with $c_{l-1} < p_l \leq c_l$
2. This experimental T_{dgemm}
   T_{dgemm} model
   α value
   → k_{dgemm_NUMA} for p_l

Automatic optimisation method

Installation phase: experimental estimation of the *SP* values

- General process: calculating the ***SP*** values that appear in the model

- $$T_{dgemm} = \frac{2n^3}{p} k_{dgemm}$$

- $$k_{dgemm} = \alpha k_{dgemm\_NUMA}(p) + (1-\alpha) k_{dgemm\_M1}$$

$$\alpha = min\left\{1, \frac{p(p-1)}{n^3 / n^2}\right\}$$

2. This experimental T_{dgemm} T_{dgemm} model α value → k_{dgemm_NUMA} for p_l

Automatic optimisation method

Installation phase: experimental estimation of the *SP* values

- General process: calculating the ***SP*** values that appear in the model
- ***SP*** values to calculate: k_{dgemm_M1} ,..., k_{dgemm_MH}
- For each memory level ***l***, from ***l*=1** until ***H***:

1. Executing `dgemm` → experimental execution time (experimental T_{dgemm}):
   - for a fixed problem size, ***n***
   - for a number of threads, p_l, with $c_{l-1} < p_l \leq c_l$

2. This experimental T_{dgemm}
   T_{dgemm} model
   α value

   → k_{dgemm_NUMA} for p_l

3. k_{dgemm_NUMA} for p_l
   k_{dgemm_NUMA} model
   values of k_{dgemm_M1} ,..., k_{dgemm_Ml-1}

   → k_{dgemm_Ml}

Automatic optimisation method

Installation phase: experimental estimation of the *SP* values

- General process: calculating the ***SP*** values that appear in the model
- ***SP*** values to calculate: k_{dgemm_M1} ,..., k_{dgemm_MH}
- For each memory level ***l***, from ***l*=1** until ***H***:

1. Executing `dgemm` → experimental execution time:
   - for a fixed problem size, ***n***
2. $$k_{dgemm\_NUMA}(p) = \frac{\sum_{l=0}^{H-2}(c_l - c_{l-1})k_{dgemm\_Ml} + (p - c_{H-1})k_{dgemm\_MH}}{p}$$

   k_{dgemm_NUMA} for p_l
3. α value

   k_{dgemm_NUMA} for p_l

   k_{dgemm_NUMA} model

   values of k_{dgemm_M1} ,..., k_{dgemm_Ml-1}

   → k_{dgemm_Ml}

Automatic optimisation method

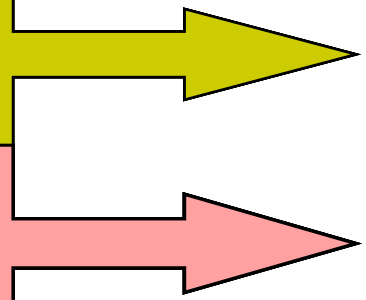
Installation phase: experimental estimation of the *SP* values

- General process: calculating the ***SP*** values that appear in the model
- ***SP*** values to calculate: k_{dgemm_M1} ,..., k_{dgemm_MH}
- For each memory level ***l***, from ***l*=1** until ***H***:

1. Executing `dgemm` → experimental execution time:
   - for a fixed (preferably small) problem size, ***n***
   - for a number of threads, p_l, with $c_{l-1} < p_l \leq c_l$

2. This experimental T_{dgemm}
   T_{dgemm} model
   α value

3. k_{dgemm_NUMA} for p_l
   k_{dgemm_NUMA} model
   values of k_{dgemm_M1} ,..., k_{dgemm_Ml-1}



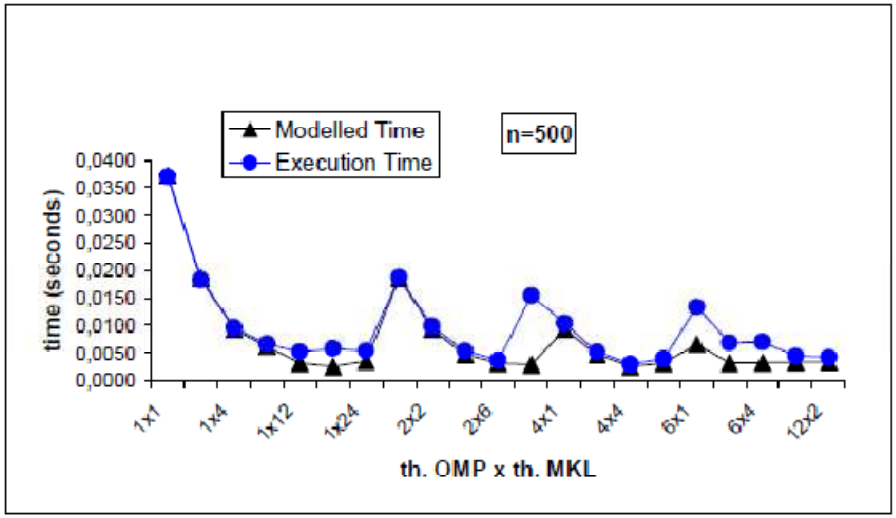
k_{dgemm_NUMA} for p_l

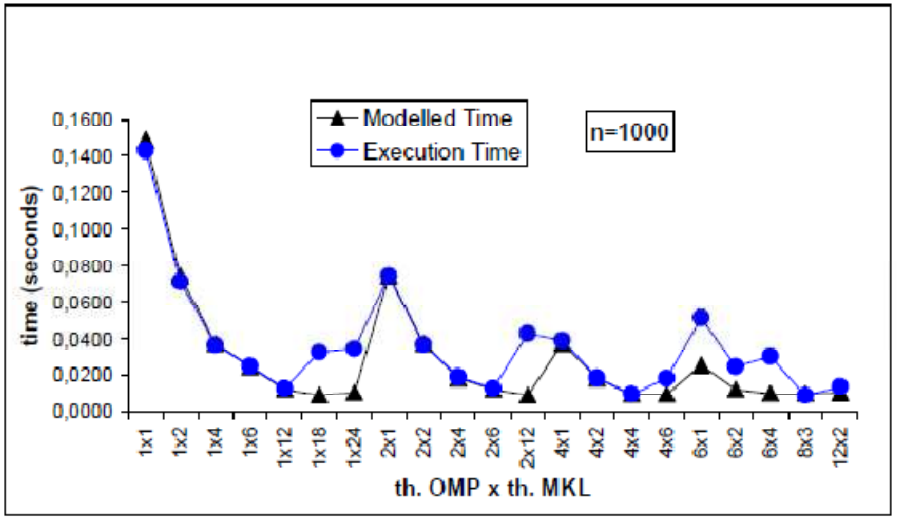
k_{dgemm_Ml}

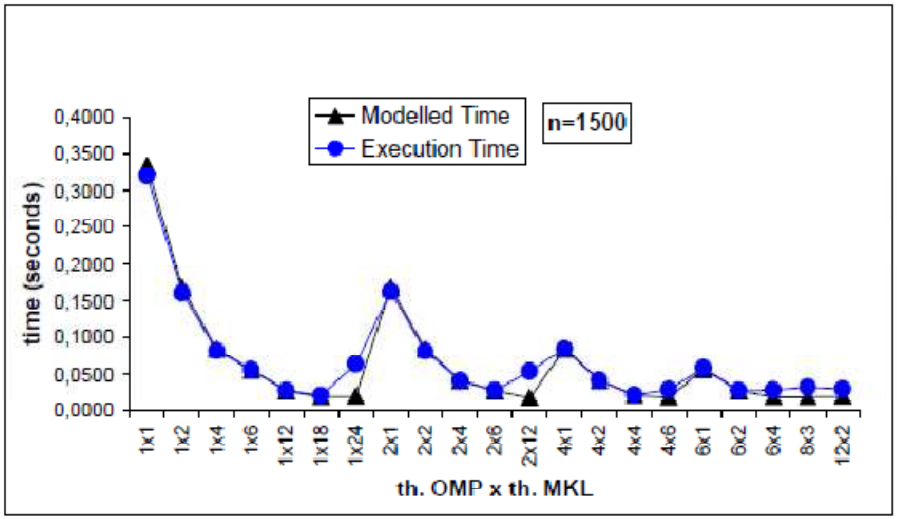
Automatic optimisation method

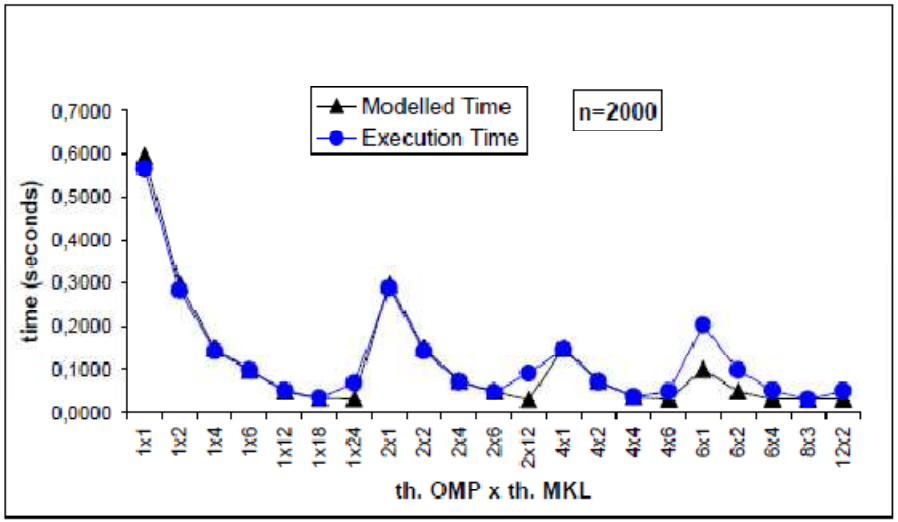
Installation phase: experimental estimation of the *SP* values

Comparison execution vs. modelled time in platform **Saturno (**`dgemm` routine)





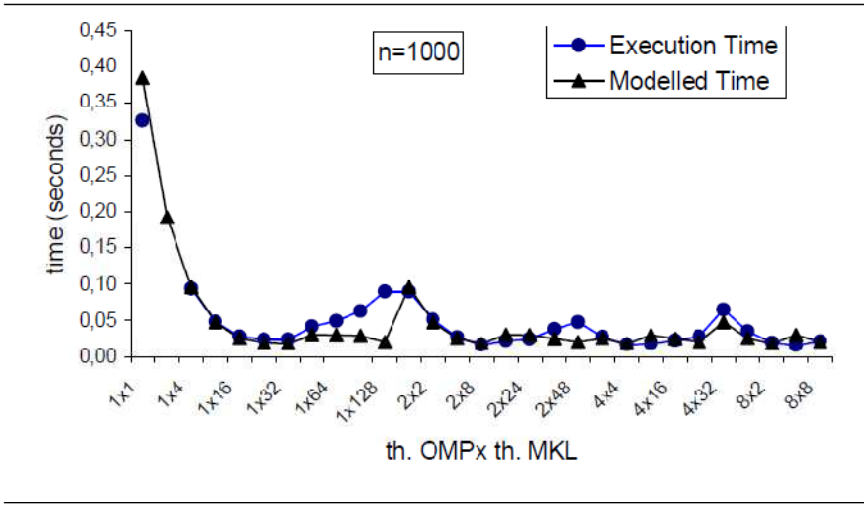


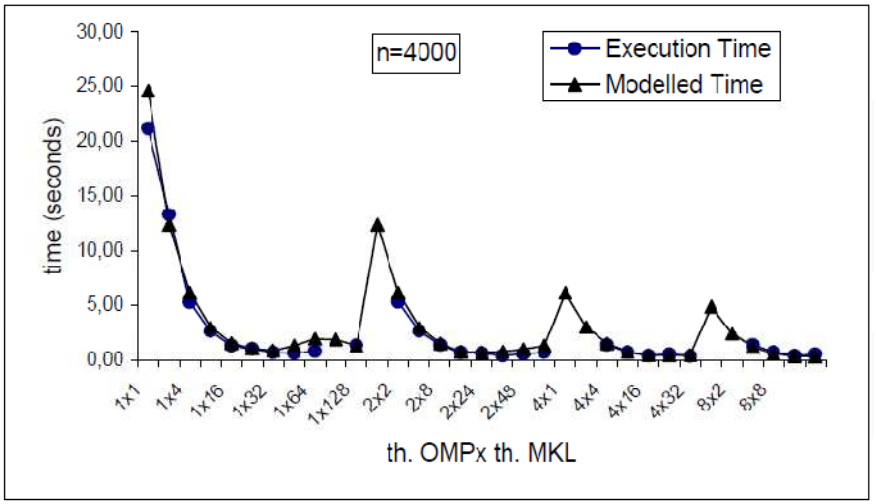


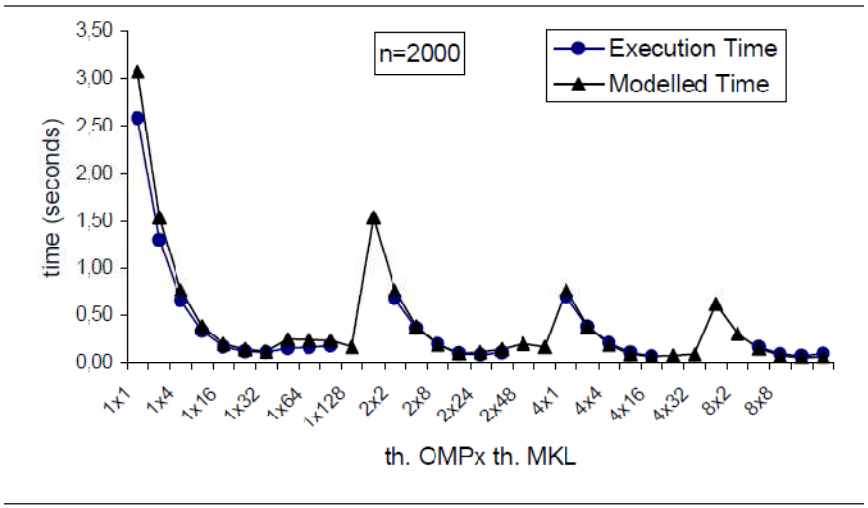
Automatic optimisation method

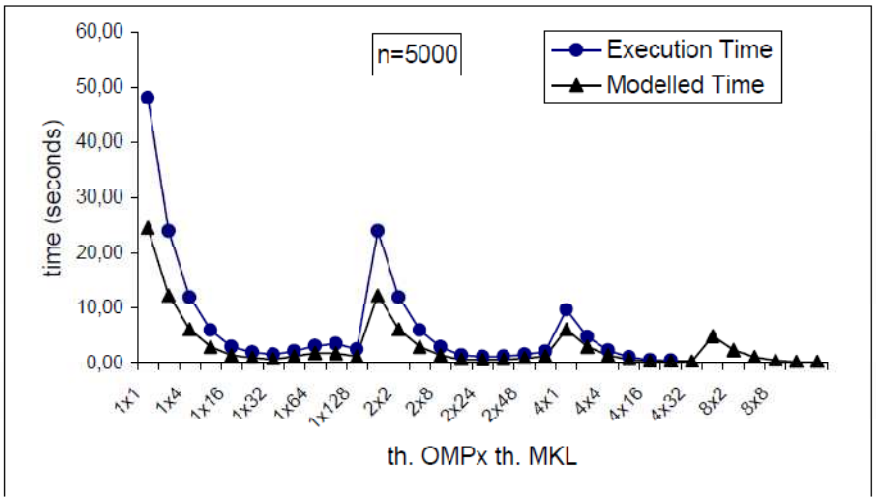
Installation phase: experimental estimation of the *SP* values

Comparison execution vs. modelled time in platform **Ben (**`dgemm` routine)





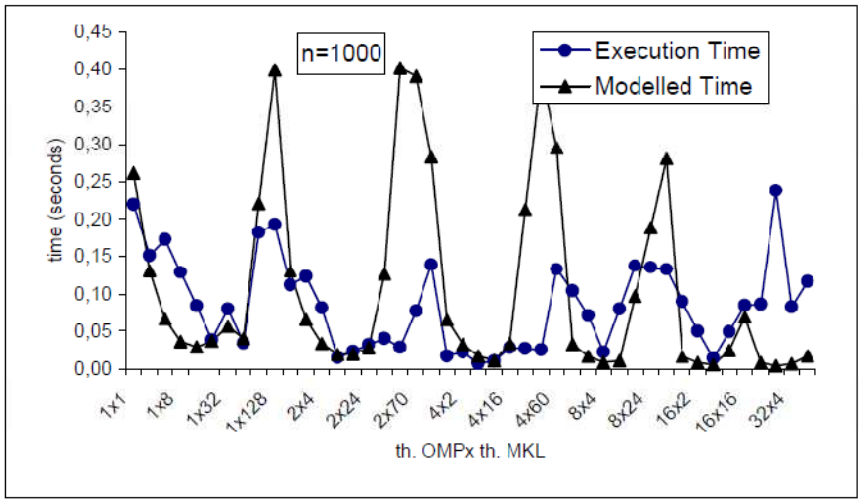


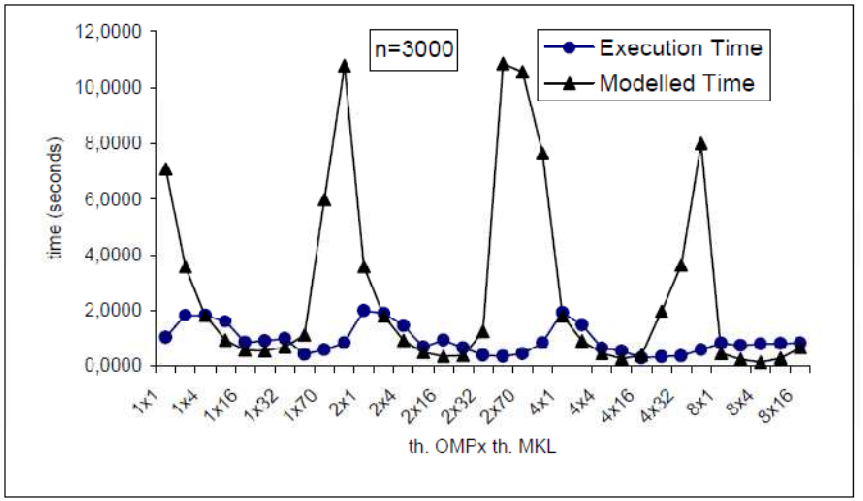


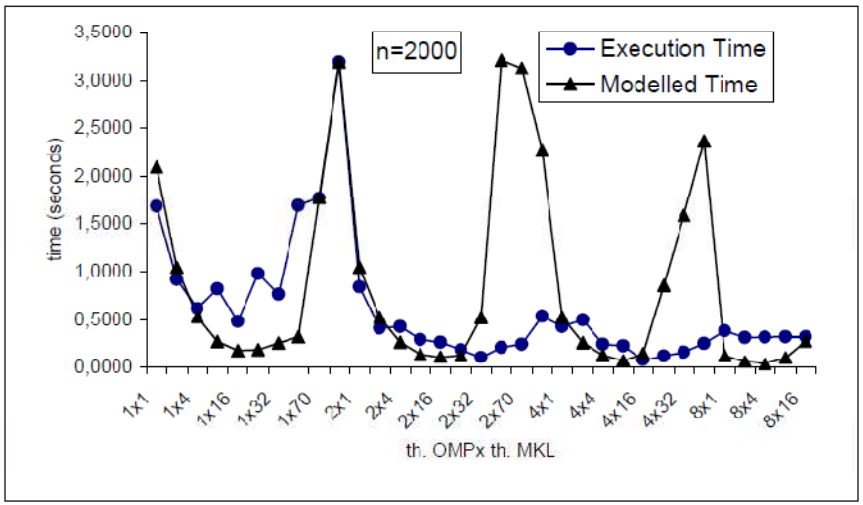
Automatic optimisation method

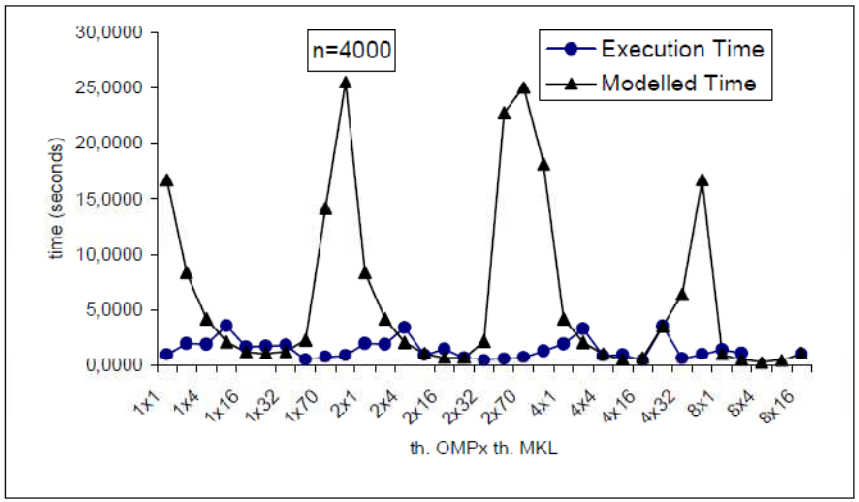
Installation phase: experimental estimation of the *SP* values

Comparison execution vs. modelled time in platform **Pirineus (**dgemm routine)









Outline

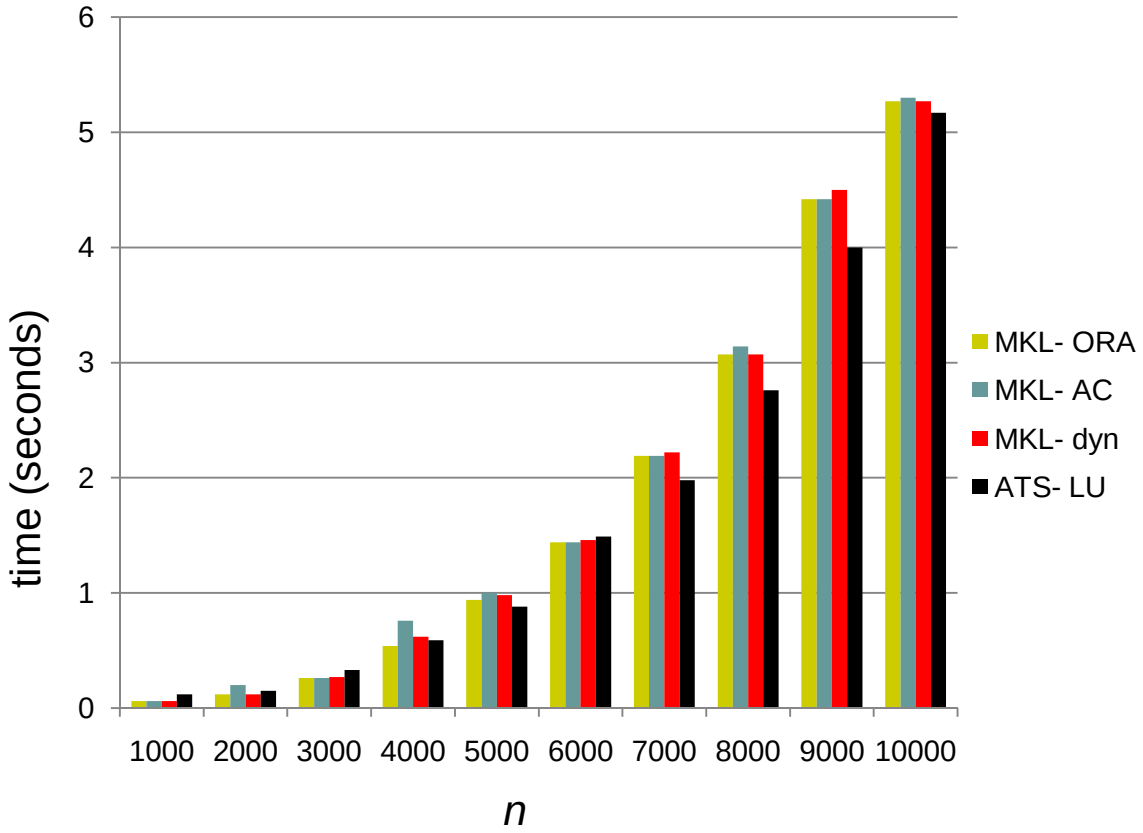
- Introduction
- Computational systems
- The software
- Motivation
- Automatic optimisation method
  - Design phase
  - Installation phase
  - Execution phase
- Conclusions and future work lines

Automatic optimisation method

Execution phase: Selection of the *AP* values

Execution time (seconds). Platform: **Saturno**

- To solve a LU factorisation problem with size ***n*** in a concrete platform:
  - The **ATS-LU:** model with ***SP*** + ***n*** → selects values for the ***AP*** (***q*** x ***p***)

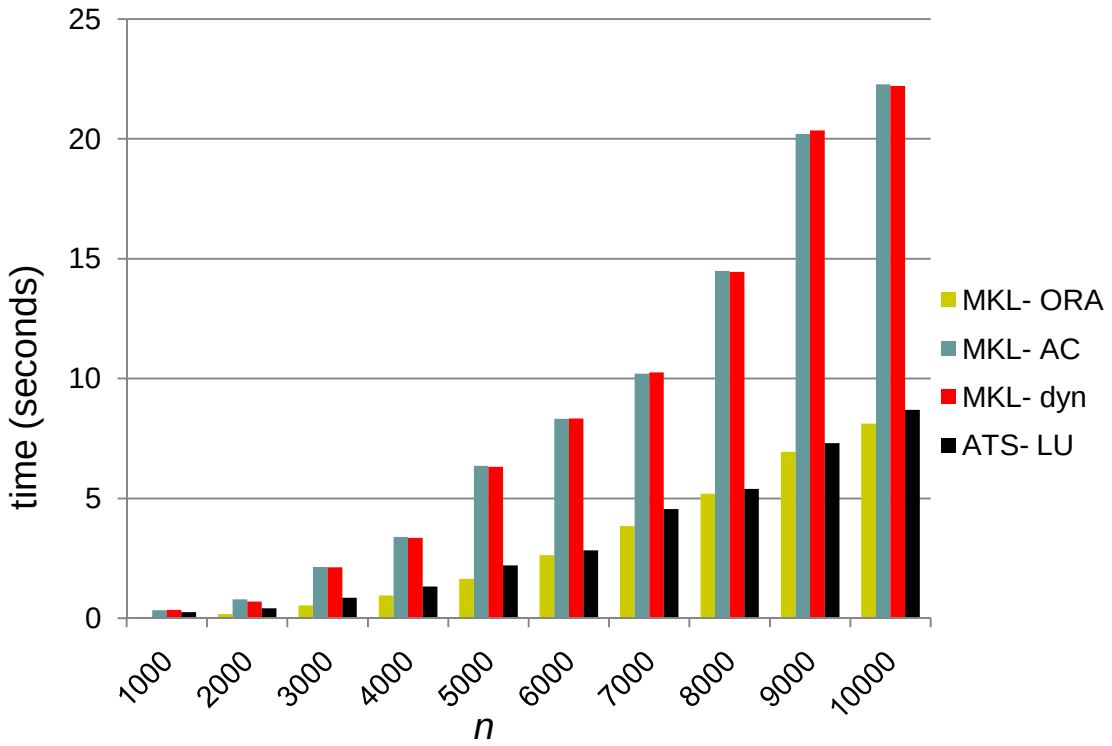


Automatic optimisation method

Execution phase: Selection of the *AP* values

Execution time (seconds). Platform: **Ben**

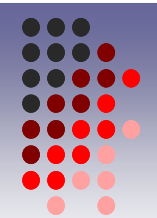
- To solve a LU factorisation problem with size ***n*** in a concrete platform:
  - The **ATS-LU:** model with ***SP*** + ***n*** → selects values for the ***AP*** (***q*** x ***p***)



Outline

- Introduction
- Computational systems
- The software
- Motivation
- Automatic optimisation method
  - Design phase
  - Installation phase
  - Execution phase
- Conclusions and future work lines

Conclusions and future work lines



- Behaviour of the kernel `dgemm` of MKL
  - Number of threads equal to number of cores: not always the best option
  - Big problems in Large Systems → OpenMP+MKL is a good option
- Reduction in the execution time of scientific codes, like LU factorisation:
  - Intensively use matrix multiplications
  - Adequately selecting the threads to be used in the solution of the problem
- Future:
  - Same methodology applied to other scientific routines
  - Different numbers of threads in different parts of the program
  - Multi-fabric libraries: routines run differently, depending on the problem