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

Jesús Cámar, Javier Cuenca, Luis P. García, Domingo Giménez

Scientific&Parallel Computing Group
University of Murcia, SPAIN
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 \texttt{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 \((q \times p\) 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

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

![Diagram showing performance characteristics for Ben and Pirineus platforms](image-url)
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

### Graphs

**Ben**

- Speed-up vs. number of OpenMP threads and MKL threads

**Pirineus**

- Speed-up vs. number of OpenMP threads and MKL threads

---

*Auto-tuned nested parallelism: a way to reduce the execution time of scientific software in NUMA systems*  
*PPAM 2012. London (U.K.)*
Outline

- Introduction
- Computational systems
- The software
- Motivation
- **Automatic optimisation method**
  - Design phase
  - Installation phase
  - Execution phase
- Conclusions and future work lines
An adaptation to large NUMA platforms:

- 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 \( \text{dgemm} \) without generating OpenMP threads

- **Model:**
  \[
  T_{dgemm} = \frac{2n^3}{p} k_{dgemm}
  \]
  \[ \text{AP: } p \rightarrow \text{number of threads inside } \text{dgemm} \]
  \[ \text{SP: } k_{dgemm} \rightarrow \text{time to perform a basic operation inside } \text{dgemm} \text{ (load\&store included)} \]
  - Taking into account NUMA and its data migration system:
    \[
    k_{dgemm} = \alpha k_{dgemm\_NUMA}(p) + (1-\alpha)k_{dgemm\_M1}
    \]
  - \( k_{dgemm\_M1} \rightarrow \text{operation time when data are in the closest memory to the core (mem. level 1)} \)
  - \( k_{dgemm\_NUMA} \rightarrow \text{operation time when data are in any level of the RAM memory} \)
  - \( \alpha \rightarrow \text{weighting factor} \)
    - 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 \( (\text{dgemm} \rightarrow n^3/n^2) \)

\[
\alpha = \min\left\{1, \frac{p(p-1)}{n^3/n^2}\right\}
\]
Automatic optimisation method

Design phase: modelling the execution time of the routine

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

- Platform:
  - $H \rightarrow$ number of memory levels
  - $c_i \rightarrow$ 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 $c_1 < p \leq c_2$:
    - $k_{dgemm_{NUMA}}(p) = \frac{c_1 k_{dgemm_{M1}} + (p - c_1) k_{dgemm_{M2}}}{p}$
  - $\ldots$, in general, if $c_{H-1} < p \leq c_H$:
    - $k_{dgemm_{NUMA}}(p) = \frac{\sum_{l=1}^{H-2} (c_l - c_{l+1}) k_{dgemm_{Ml}} + (p - c_{H-1}) k_{dgemm_{MH}}}{p}$
Automatic optimisation method
Design phase: modelling the execution time of the routine
Modelling 2-Level: OpenMP threads + MKL multithreading \texttt{dgemm}

- Model:
  \[
  T_{2L_{dgemm}} = \frac{2^n}{q} \left( \frac{nn}{p} k_{2L_{dgemm}} + \frac{2n^3}{k_{2L_{dgemm}}} \right)
  \]

- \texttt{AP} \rightarrow R = p \times q \text{ threads} interactuating
  - \texttt{p} → Number of threads inside the MKL routine \texttt{dgemm}
  - \texttt{q} → Number of OpenMP threads

- \texttt{SP} \rightarrow k_{2L_{dgemm}}: time to carry out a basic operation
  \[
  k_{2L_{dgemm}} = \alpha k_{2L_{dgemmNUMA}}(R, p) + (1 - \alpha) k_{2L_{dgemmM1}}
  \]
  \[
  k_{2L_{dgemmNUMA}}(R, p) = \frac{k_{dgemmNUMA}(R) + k_{dgemmNUMA}(p)}{2}
  \]

  \[
  \alpha = \min \left\{ 1, \frac{R(R - 1)}{n^3} \right\}
  \]
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: $k_{dgemm_M1}, \ldots, k_{dgemm_MH}$

- For each memory level $l$, from $l=1$ until $H$:
  1. Executing $\text{dgemm}$ → experimental execution time (experimental $T_{\text{dgemm}}$):
     - 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

- \textit{SP} values to calculate: $k_{\text{dgemm}_M}, \ldots, k_{\text{dgemm}_M^H}$

- For each memory level $l$, from $l=1$ until $H$:
  
  1. Executing $\text{dgemm} \rightarrow$ experimental execution time (experimental $T_{\text{dgemm}}$):
     - for a fixed problem size, $n$
     - for a number of threads, $p_i$, with $c_{i-1} < p_i \leq c_i$
  
  2. This experimental $T_{\text{dgemm}}$
     
     $T_{\text{dgemm}}$ model
     
     $\alpha$ value
     
     $k_{\text{dgemm}_\text{NUMA}}$ for $p_i$
Automatic optimisation method
Installation phase: experimental estimation of the $SP$ values

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

\[
T_{\text{dgemm}} = \frac{2n^3}{p} k_{\text{dgemm}}
\]

\[
k_{\text{dgemm}} = \alpha k_{\text{dgemm[NUMA]}(p)} + (1 - \alpha) k_{\text{dgemm[M1]}}
\]

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

2. This experimental $T_{\text{dgemm}}$

$k_{\text{dgemm[NUMA]}}$ 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: $k_{dgemm_M1}, \ldots, k_{dgemm_MH}$

- For each memory level $l$, from $l=1$ until $H$:
  1. Executing $dgemm \rightarrow$ experimental execution time (experimental $T_{dgemm}$):
     - for a fixed problem size, $n$
     - for a number of threads, $p$, with $c_{l-1} < p \leq c_{l}$

2. This experimental $T_{dgemm}$ model $\alpha$ value

3. $k_{dgemm_{NUMA}}$ for $p_l$
   $k_{dgemm_{NUMA}}$ model values of $k_{dgemm_M1}, \ldots, k_{dgemm_Ml-1}$

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

PPAM 2012. London (U.K.)
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: \( k_{\text{dgemm}_M1}, \ldots, k_{\text{dgemm}_MH} \)
- For each memory level \( l \), from \( l=1 \) until \( H \):
  1. Executing \( \text{dgemm} \rightarrow \) experimental execution time:
     - for a fixed problem size, \( n \)
  2. \[
k_{\text{dgemm_NUMA}}(p) = \sum_{l=0}^{H-2} (c_l - c_{l-1}) k_{\text{dgemm}_Ml} + (p - c_{H-1}) k_{\text{dgemm}_MH} \]
  3. \( k_{\text{dgemm_NUMA}} \) for \( p_l \)

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

PPAM 2012. London (U.K.)
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: \( k_{dgemm_M1}, \ldots, k_{dgemm_MH} \)

- 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 \( T_{dgemm} \) model
     - \( T_{dgemm} \)
     - \( \alpha \)

  3. \( k_{dgemm_{NUMA}} \) for \( p_l \)
     - \( k_{dgemm_{NUMA}} \)
     - \( k_{dgemm_{NUMA}} \) model
     - values of \( k_{dgemm_M1}, \ldots, k_{dgemm_Ml-1} \)
Automatic optimisation method

Installation phase: experimental estimation of the $SP$ values

Comparison execution vs. modelled time in platform **Saturno** ($\text{dge}_{\text{mm}}$ routine)

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

PPAM 2012. London (U.K.)
Automatic optimisation method

Installation phase: experimental estimation of the $SP$ values

Comparison execution vs. modelled time in platform Ben ($\text{dgemm routine}$)
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 $SP + n \rightarrow$ selects values for the AP $(q \times p)$
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 $SP + n \rightarrow$ selects values for the $AP \ (q \times p)$
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