Static versus Dynamic Memory Allocation: a Comparison for Linear Algebra Kernels

Toufik Baroudi[1] Vincent Loechner[2] Rachid Seghir[1]

[1]University of Batna, Algeria

[2]University of Strasbourg & INRIA, France

IMPACT 2020

Introduction

Two years ago [BSL17, TACO]:

- compact data layout for regular sparse matrices
- optimized by Pluto
- our preliminary benchmarks were inconsistent

$\rightarrow$ due to matrix allocation mode: as **static declared array** or as **array of pointers** to dynamically allocated memory

Introduction: Content of this Presentation

We precisely analyze one code: triangular matrix multiplication

- using the performance counters (#instr., #mem. access, #L1-L3 cache misses, #TLB misses, #vectorized instr.)

Ran the same tests on the PolyBench linear algebra kernels

Introduction: Objective

Array allocation mode influences performance!

Main factors of performance variation:

- ability of the compiler to detect **vectorization**
- number of **cache misses** and **memory loads**

Introduction: Objective

Array allocation mode influences performance!

Main factors of performance variation:

- ability of the compiler to detect **vectorization**
- number of **cache misses** and **memory loads**

This work is not a manifest for one type of allocation or the other,
it is a warning: declaration and allocation of arrays matters!

Comparing various versions of codes using different array allocation modes can get biased

Table of Contents

Triangular Matrix Multiplication

Demo

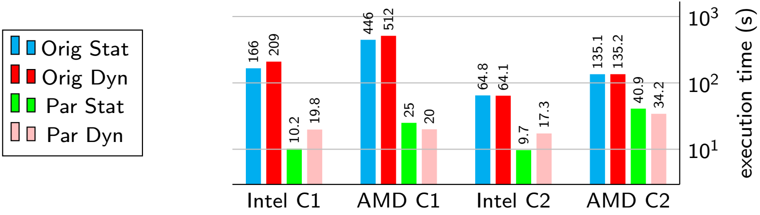
in completely different conditions than in the paper:
on this laptop (MacOS 10.14, clang/llvm-9.0.0, 4-cores Intel core i7)

Table of Contents

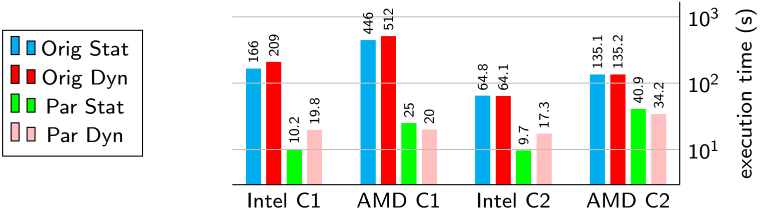
Triangular Matrix Multiplication: setup

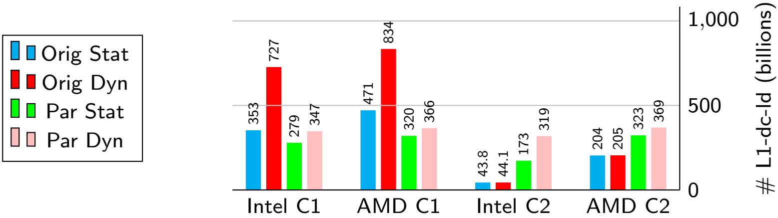
- Intel platform: dual socket Intel Xeon E5-2650v3 (Haswell-EP)
  2x10 hyperthreaded cores, AVX2 (256 bits)
- AMD platform: dual socket AMD Opteron 6172 (Magny-Cours)
  2x12 cores, SSE (128 bits)
- using `pluto-0.11.4 --tile --parallel`
- using `gcc-7.4.0 -O3 -march=native -fopenmp`
- on a regular `Linux 4.0.15 (Ubuntu)`
- problem size: `N=8000`

Triangular Matrix Multiplication: execution time

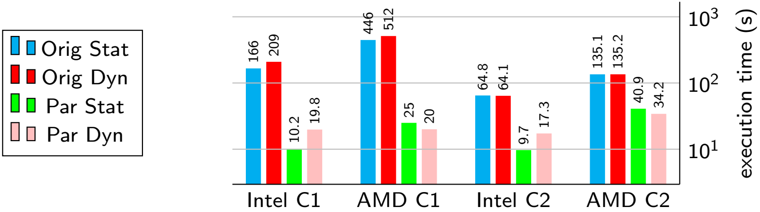


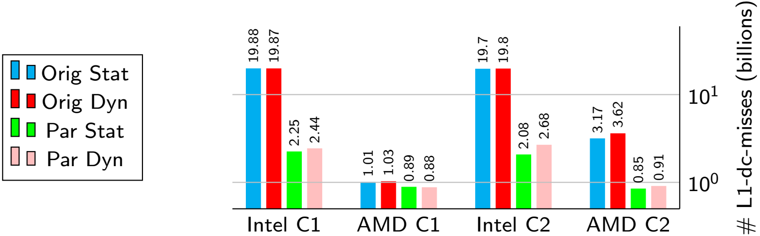
Triangular Matrix Multiplication: L1-dcache-loads



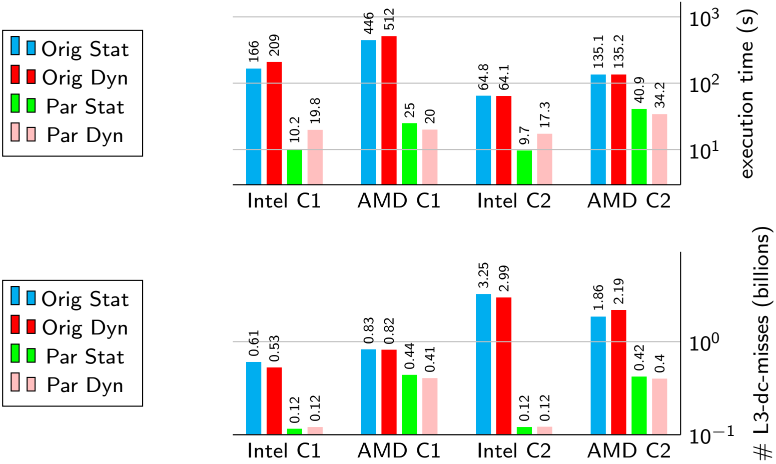


Triangular Matrix Multiplication: L1-dcache-misses

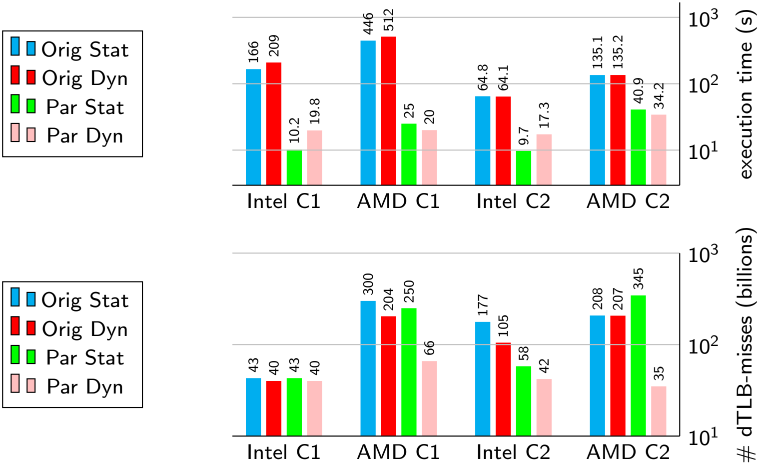




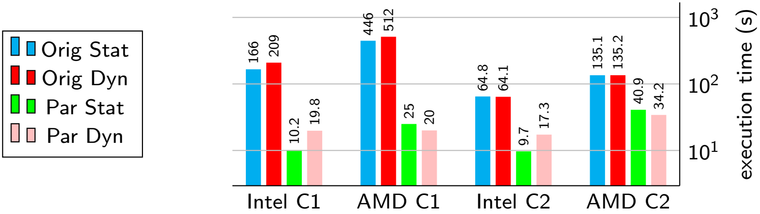
Triangular Matrix Multiplication: L3-dcache-misses

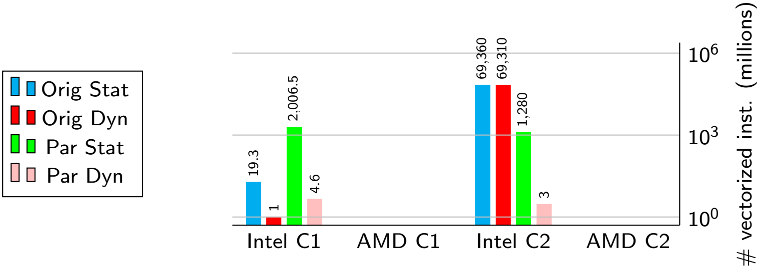


Triangular Matrix Multiplication: dTLB-misses



Triangular Matrix Multiplication: vectorized instructions





unavailable on the AMD but "`gcc -fopt-info-vec`" seems to confirm the correlation

Triangular Matrix Multiplication: Synthesis

- array allocation mode has a significant impact on the performance of this code
- it can have opposite effects on different processors!
- factors of influence:
  - number of memory accesses
  - number of cache and TLB misses
  - number of **vectorized instructions**
- other experiments[1] on the Intel platform show that the number of vectorized instructions is a major factor of influence

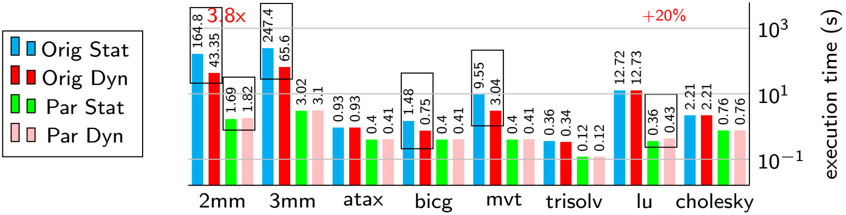
[1] on other triangular matrix kernels: Cholesky, SolveMat, sspfa.

Table of Contents

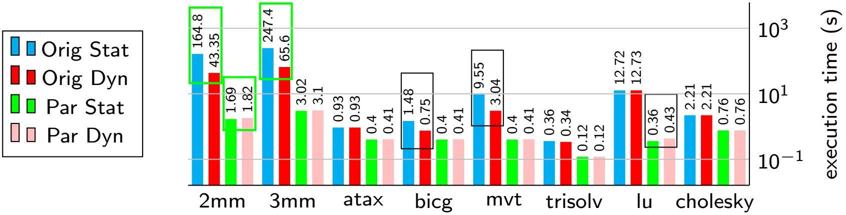
PolyBench: setup

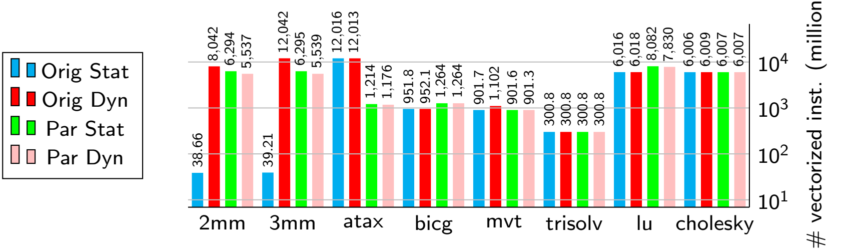
- on the Intel platform
- using `pluto-0.11.4 --tile --parallel`
- using `gcc-7.4.0 -O3 -march=native -fopenmp`
- PolyBench macro `POLYBENCH_STACK_ARRAYS`:
  - static version: stack allocated static array
  - dynamic version: multidimensional heap-allocated array (**not an array of pointers** as in the previous experiment)
- problem size:
  - N=2,000 for $O(N^3)$ algorithms
  - N=20,000 for $O(N^2)$ algorithms

PolyBench: execution time

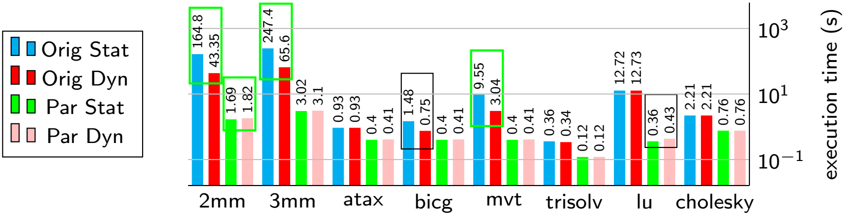


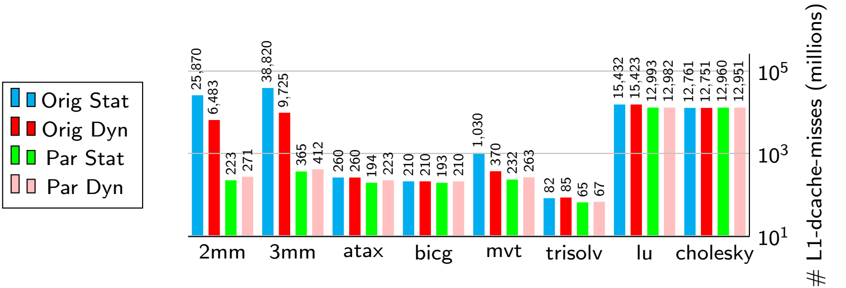
PolyBench: vectorized instructions



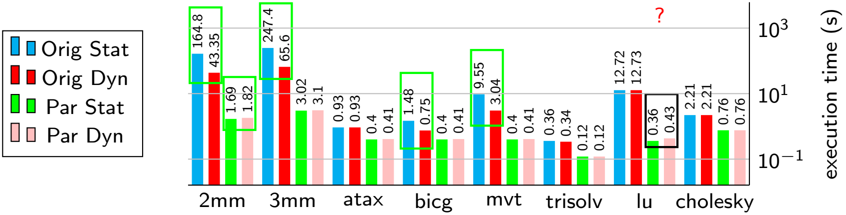


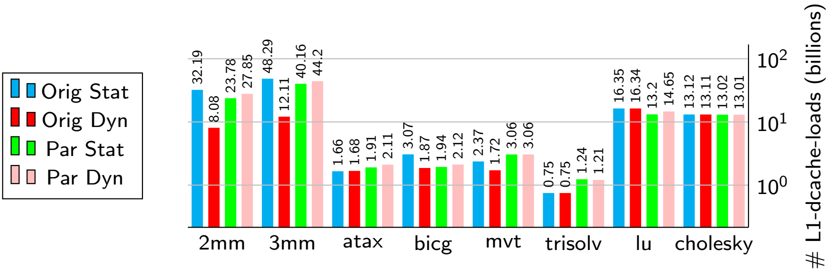
PolyBench: L1 cache misses



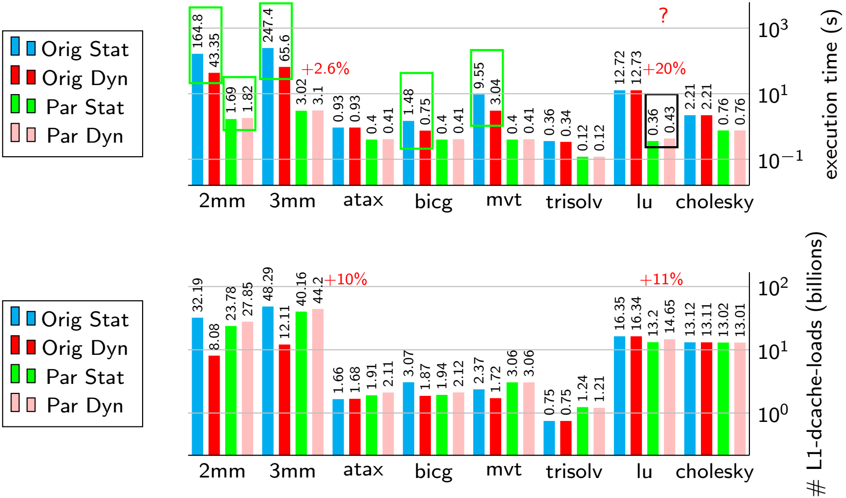


PolyBench: memory loads





PolyBench: memory loads



Analysis

- factors of influence:
  - number of vectorized instructions
  - number of cache misses
  - number of memory accesses?
  - ?
- why are there more/less memory accesses when allocating arrays statically or dynamically?

  → we suspect that the varying pressure on register allocation changes the compiler's decision on data reuse (*bicg*)

Table of Contents

Conclusion

- Array allocation has a **significant impact** on performance in many cases
- It can be alternatively in favor of static or dynamic allocation and it can even flip on different architectures!
- Be careful when you compare codes using different allocations (e.g. when working on data layout transformations): this side effect **could bias your measurements**!