Memory Performance Comparison of Heap and Data Segments with Different Compiler Optimizations

Bruno Oliveira Cattelan, **Lucas Mello Schnorr**
{bocattelan,schnorr}@inf.ufrgs.br

Porto Alegre, Septemper 2016 — 2^{nd} WSPPD

Outline

Challenge

When I first started working with professor Schnorr we were asked to develop the fastest matrix multiplication algorithm that we could
The ideia was for us to:

- Be introduced to parallel computing
- Learn how to use the OpenMP library

Memory Allocation

In about the same time I had a class about memory management that got me wondering

- The data segment is really simple to address
- The heap has more levels of indirection

Question Can these differences have an impact in the execution time of a program?

The Heap and the Cache

In the book [**High Performance Memory Systems**] is shown that the heap has a higher cache-miss rate than other types of memory allocation

First Tests

I made two programs that multiplied two matrices

- One that allocated the matrices in the data segment
- One that allocated the matrices in the heap

Tests

Using the DoE package from R we devised the experiments using a full factorial approach with the following factors:

- Size
- Optimization
- Algorithm (Memory Allocation)

Machine

The tests were run in the following machine:

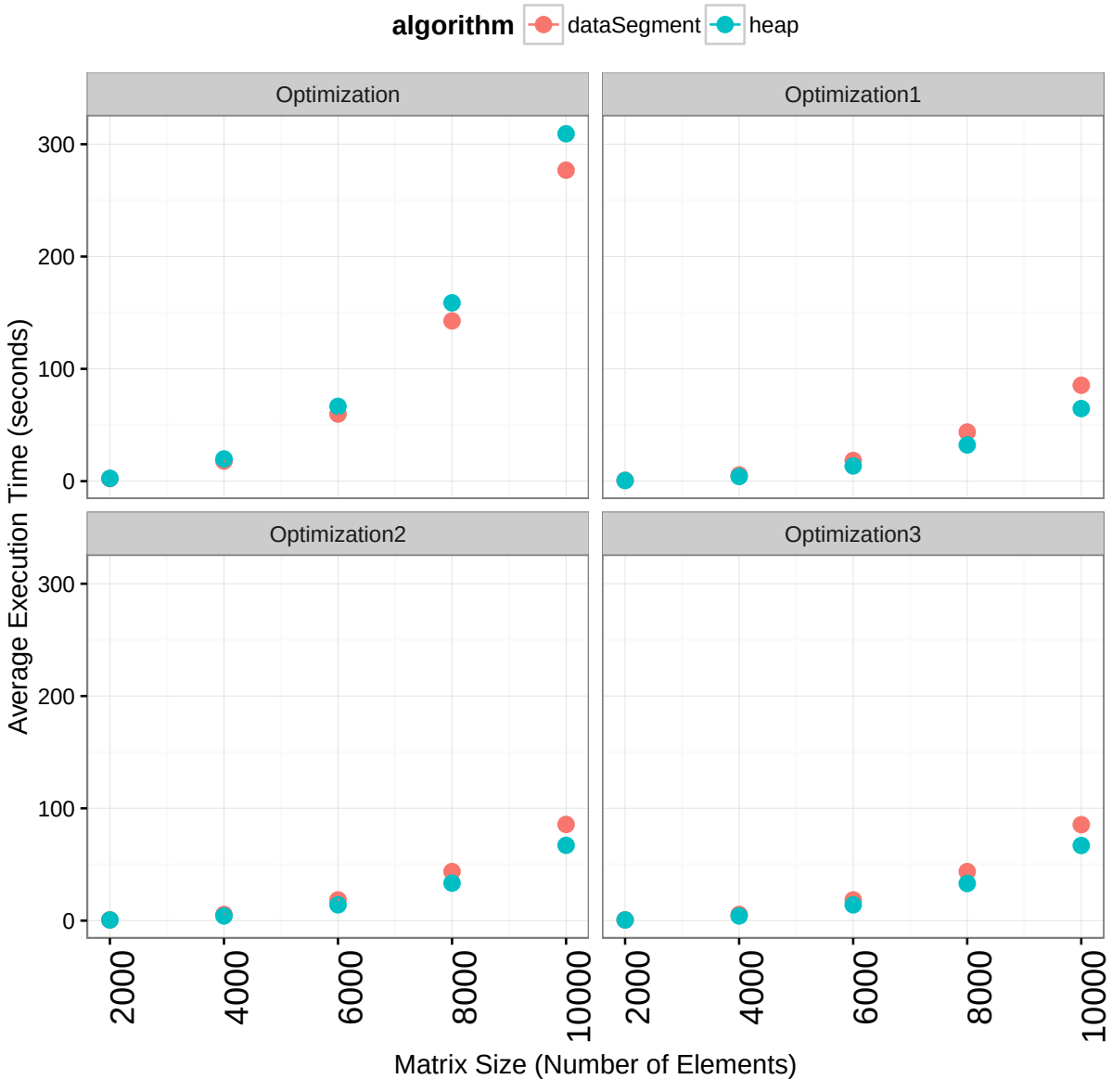
Architecture:	$x86_{64}$
CPU(s):	32
Thread(s) per core:	2
Core(s) per socket:	8
Socket(s):	2
NUMA node(s):	2
CPU MHz:	1202.968
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	20480K
NUMA node0 CPU(s):	0-7,16-23
NUMA node1 CPU(s):	8-15,24-31

NUMA Effect

To prevent any possible interference by the NUMA memory, all tests were executed using:

- $\text{GOMP}_{\text{CPUAFFINITY}}$=0-31
  - It prevents the OS from moving the threads between the processors
- numactl -i all
  - It makes all alocations follow a round-robin policy, rather than by proximity

Matrix Multiplication Results

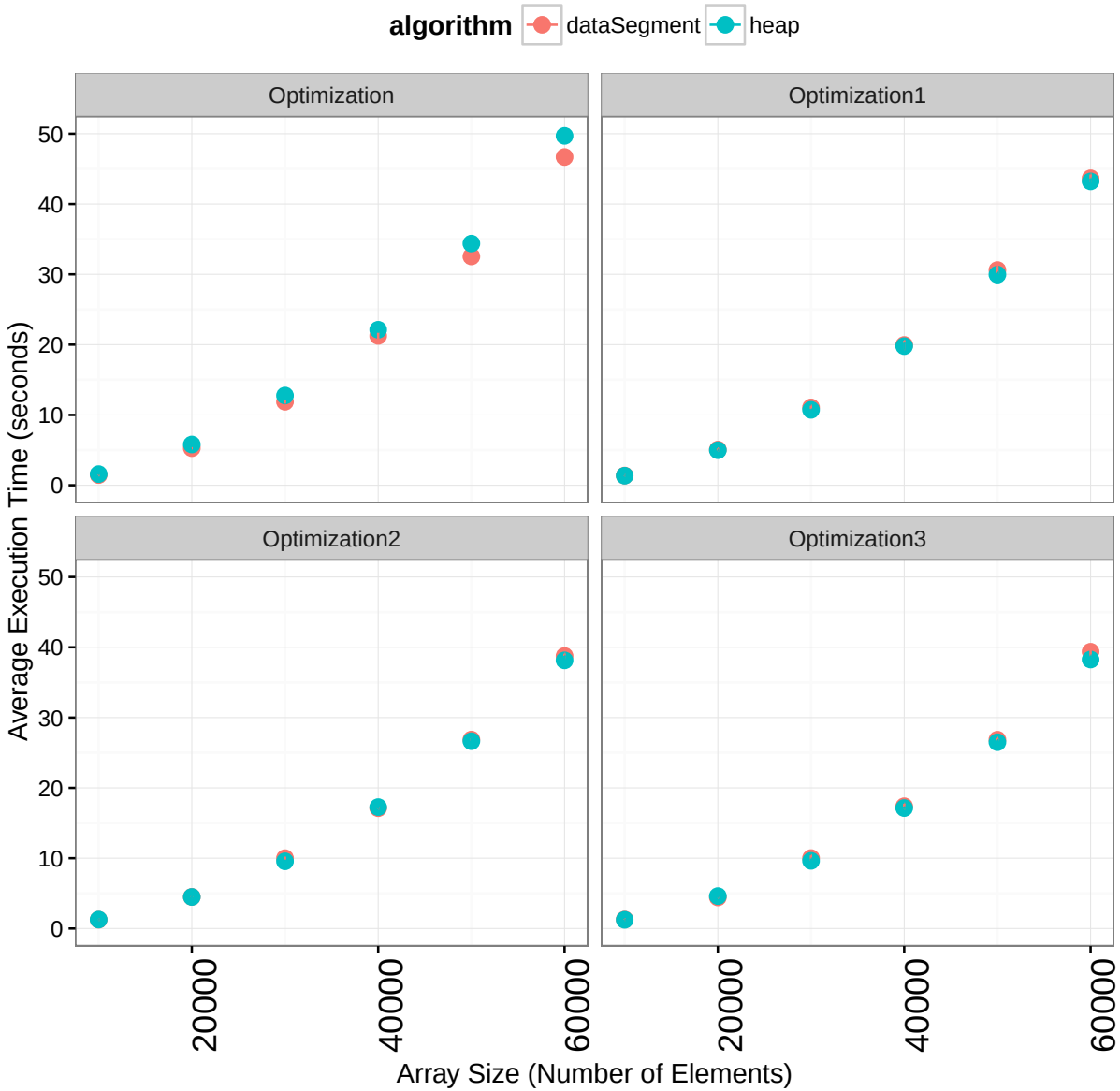


Array

As the Matrix Multiplication was too complex to analyze, I decided to use a simpler program

- It just added to an array the index divided by the array size to the current variable the array size times

Array Results



Problem

The whole array fit inside the cache
But the behavior was the same

New Idea

For no optimization the data segment is faster than the heap
GCC can optimize better the heap than the data segment

Assembly - No Optimization

Data Segment

```
.L6:
  movq  -40(%rbp), %rax
  movsd  bigArray(,%rax,8),
```

Heap

```
.L6:
  movq  bigArray(%rip), %rax
  movq  -40(%rbp), %rdx
  salq  $3, %rdx
  leaq  (%rax,%rdx), %rcx
  movq  bigArray(%rip), %rax
  movq  -40(%rbp), %rdx
  salq  $3, %rdx
  addq  %rdx, %rax
  movsd  (%rax), %xmm1
```

The heap has much more instructions than the data segment

Assembly - 1 Optimization Level

Data Segment

```
.L8:
  movq  %rcx, %rax
  movl  $0, %edx
  divq  %rbx
  testq  %rax, %rax
  js  .L4
  cvtsi2sdq  %rax, %xmm0
  jmp  .L5
```

Heap

```
.L8:
  movq  bigArray(%rip), %rax
  leaq  (%rax,%rcx,8), %rsi
  movq  %rcx, %rax
  movl  $0, %edx
  divq  %rbx
  testq  %rax, %rax
  js  .L4
  cvtsi2sdq  %rax, %xmm0
  jmp  .L5
```

Even though the heap has more instructions, it shows a lower execution time

Assembly - 1 Optimization Level More Differences

Data Segment

```
.L5:
  addsd  bigArray(,%rcx,8), 
  movsd  %xmm0, bigArray(,%r
  addq  $1, %rcx
  cmpq  %rsi, %rcx
  jne  .L8
.L7:
  addq  $1, %rdi
  cmpq  %rdi, %rbx
  jbe  .L2
```

Heap

```
.L5:
  addsd  (%rsi), %xmm0
  movsd  %xmm0, (%rsi)
  addq  $1, %rcx
  cmpq  %rdi, %rcx
  jb  .L8
.L7:
  addq  $1, %r8
  cmpq  %r8, %rbx
  jbe  .L2
```

Conclusion

- For no optimization the data segment is faster
- GCC can optimize the heap in a more efficient way

Future Work

- Use a trace tool to further study the behavior
- Study the stack