Hierarquia de Memória:
Impacto dos Algoritmos / Codificação

Arquitetura de Computadores

Lic. em Engenharia Informática

João Luís Sobral

Hierarquia da Memória: Algoritmos

Conteúdos	**2 – Hierarquia da Memória**
	2.6 – Algoritmos “amigáveis” da hierarquia de memória
	2.7 – Modelos de complexidade dos algoritmos para a hierarquia de memória
Resultados de Aprendizagem	R2.4 – Identificar o **impacto** da hierarquia de memória no desempenho de programas escritos **em linguagens de alto nível**
	R2.5 - Aplicar técnicas **de optimização de programas para a hierarquia de memória**

Algoritmos “amigáveis” da hierarquia de memória

Como prever/comparar desempenho de algoritmos?

Abordagem tradicional: notação “big O”:

- número de operações efectuadas em função da dimensão dos dados

- inserção numa lista ligada: O(n), n = elementos na lista
- inserção numa árvore binária: O($\log_2(n)$)
- inserção numa tabela de *hash*: O(1)

Será este modelo adequado para prever o desempenho quando

$$\#CC_{MEM} >> CC_{CPU} \; ?$$

$$T_{exec} = (\#CC_{CPU} + \#CC_{MEM}) * T_{cc}$$

- A notação “big O” está essencialmente relacionada com #I,
não considerando os acessos à memória

$$Texe \cong \#CC_{CPU} * T_{cc} = \#I * CPI_{CPU} * T_{cc}$$

Algoritmos “amigáveis” da hierarquia de memória

Existem classes de algoritmos cujo desempenho está limitado pelo desempenho da memória (“memory bound”)

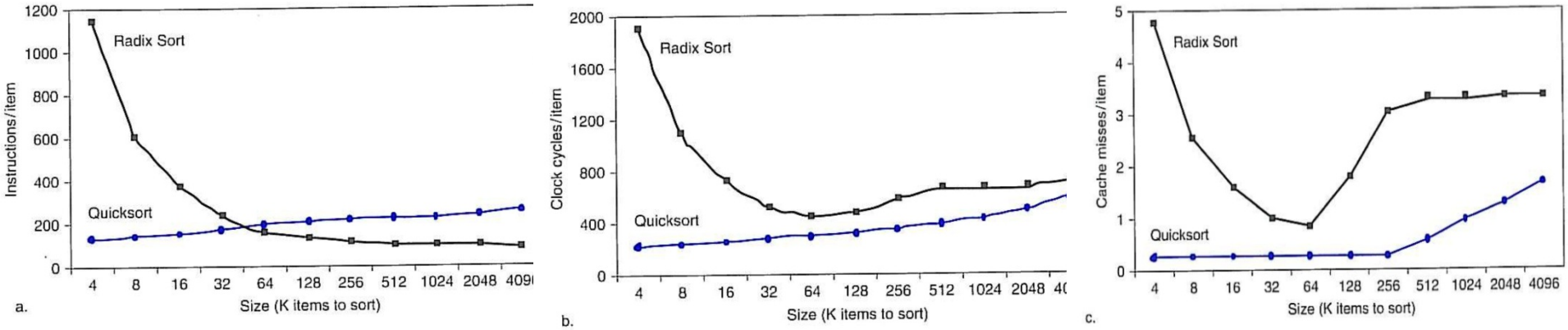
- nesse tipo de algoritmos $\#CC_{MEM}$ é muito superior a $\#CC_{CPU}$
- assim, o melhor algoritmo é aquele que minimiza o $\#CC_{MEM}$

$$\#CC_{MEM} = n^{o}\, de\ misses * miss\ penalty$$

Ou seja, aquele que apresenta menor número de *misses*

Exemplo: algoritmos de ordenação se números:

Quicksort – O (nlog(n)) *versus* radix sort – O(nk)



Modelo de complexidade dos algoritmos considerando a hierarquia de memória

Que modelo utilizar para comparar algoritmos “memory bound”?

- O modelo tradicional de complexidade não considera o impacto da hierarquia de memória:
  - é baseado num modelo RAM (*Random Access Machine*), onde a memória é considerada “plana” e com um tempo de acesso uniforme

Modelo de memória externa *(*modelo mais popular para comparar algoritmos considerando a hierarquia de memória*)*:

- Considera dois níveis de memória: **externa** e **interna** (*cache*)
- A memória **externa** é **dividida** em **blocos** de tamanho B
- A **complexidade** do algoritmo é expressa em termos do **nº de blocos transferidos** entre os dois níveis de memória (ou seja, o número de *cache misses / writeback*)

Modelo de complexidade dos algoritmos considerando a hierarquia de memória

Modelo de memória externa

exemplo: somar um vetor de N elementos contíguos na memória:

```
1  int sumvec(int v[N])
2  {
3      int i, sum = 0;
4
5      for (i = 0; i < N; i++)
6          sum += v[i];
7      return sum;
8  }
```

implica a transferência de N/B blocos entre a memória e a *cache*

a complexidade é O(N/B)

para o caso de blocos de 16 bytes (4 inteiros de 32 bits)

teremos N/4 misses:

v[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$
Access order, [h]it or [m]iss	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]

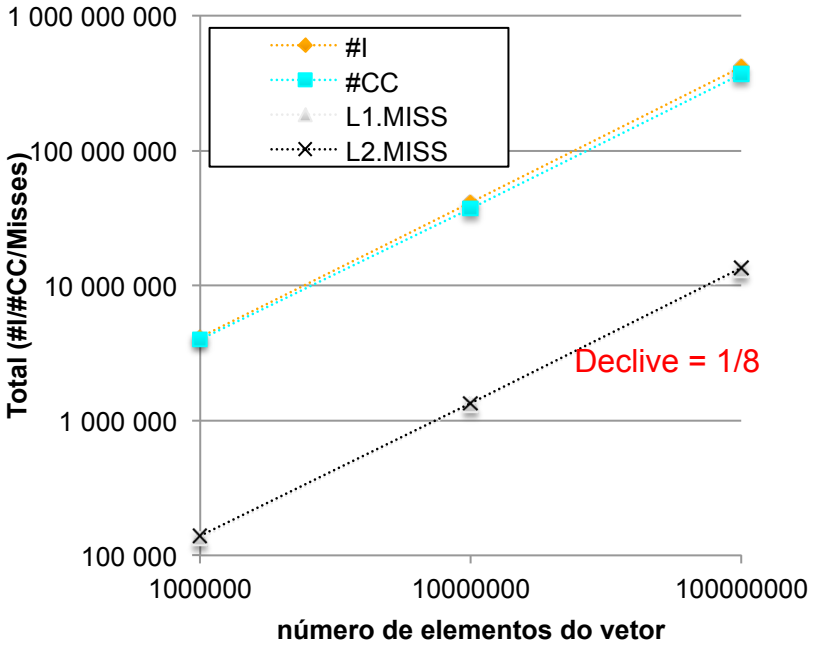
Modelo de complexidade dos algoritmos considerando a hierarquia de memória

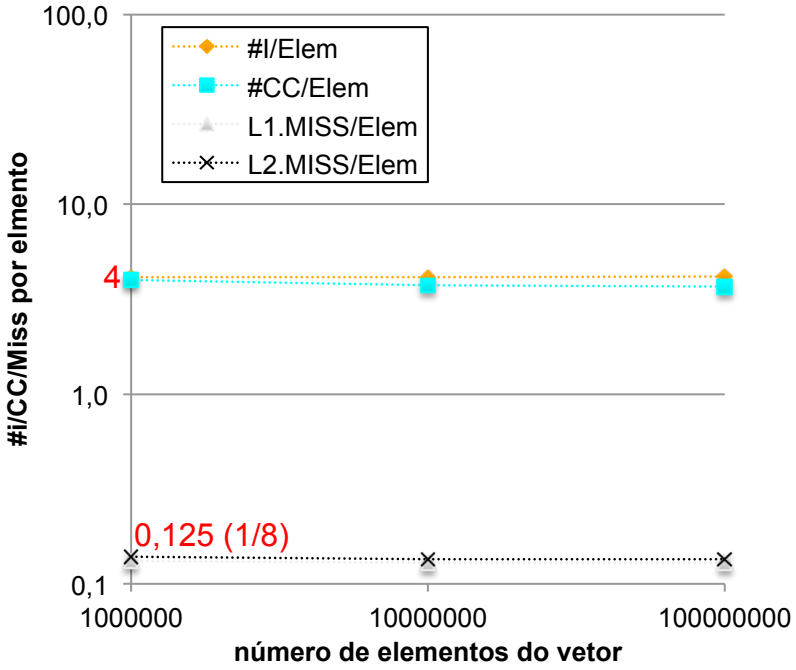
Resultados: somar vetor de *doubles* (8 bytes/Elem, 64bytes por linha de cache)

```
double sum=0;

for(int i=0; i<N; i++)
      sum += v[i];
```

```
            xorl    %eax, %eax
            xorpd   %xmm0, %xmm0
_sum:       addsd   (%rdx,%rax,8), %xmm0
            addq    $1, %rax
            cmpl    %eax, %ecx
            jg      _sum
```





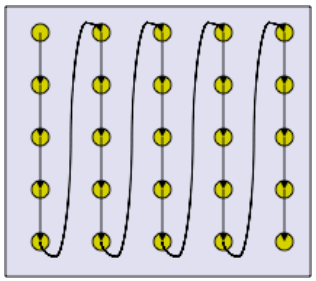
Algoritmos “amigáveis” da hierarquia de memória

Como tornar a implementação de um algoritmo mais amigável da “cache”?

Minimizar o número de blocos transferidos entre os vários níveis de memória (ou seja, o número de *misses*)

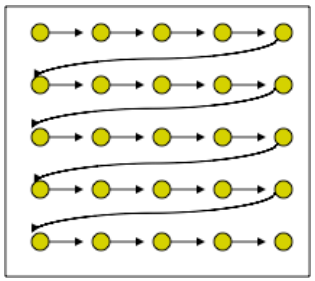
Reorganização do código/dados para melhorar a localidade espacial

- Reorganização dos blocos de código
  - Organizar os blocos na memória pela ordem que são executados
- Reorganizar os acessos aos dados
  - Aceder aos elementos de dados pela ordem que estão armazenados



Antes	Depois

for (j = 0; j < 100; j = j+1)
  for (i = 0; i < 5000; i = i+1)
  x[i][j] = 2 * x[i][j];
	for (i = 0; i < 5000; i = i+1)
	  for (j = 0; j < 100; j = j+1)
	  x[i][j] = 2 * x[i][j];



Transfere 5000*100 Blocos

Transfere 5000*100/B Blocos

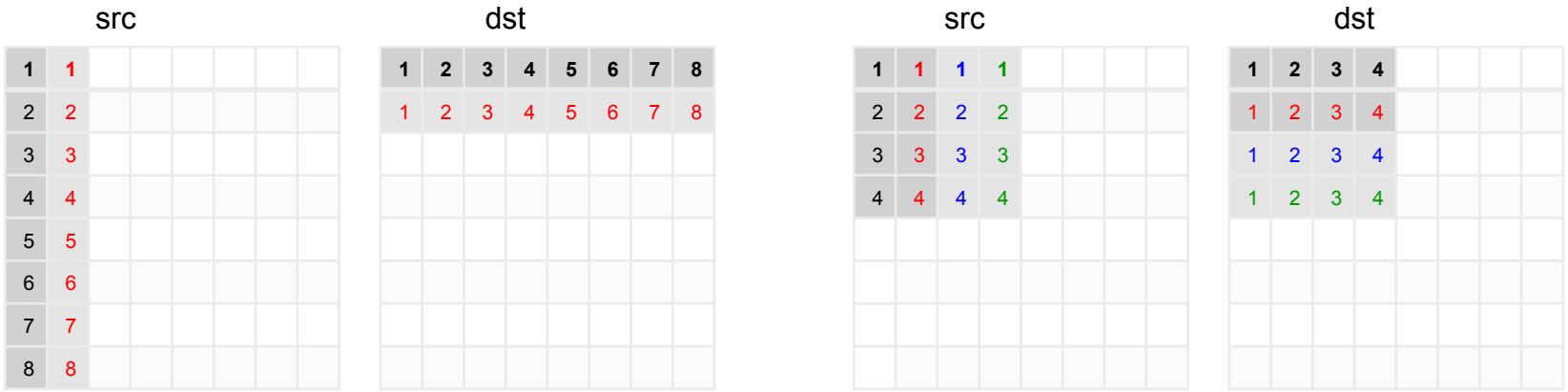
Algoritmos “amigáveis” da hierarquia de memória

Processar os dados por blocos (que cabem na *cache*) para melhorar a localidade espacial/temporal no acesso aos dados:

- Aceder / armazenar os dados em blocos, exemplo: transpor uma matriz

Antes	Depois

for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  dst[i][j] = src[j][i];
	for (int ii = 0; ii < N; ii += blocksize)
	  for (int jj = 0; jj < N; jj += blocksize)
	  // transpose the block beginning at [ii,ji]
	  for (int i = ii; i < ii + blocksize; i++)
	  for (int j = jj; j < jj + blocksize; j++)
	  dst[i][j] = src[j][i];

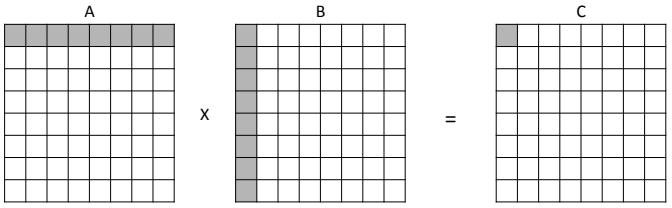


Algoritmos “amigáveis” da hierarquia de memória

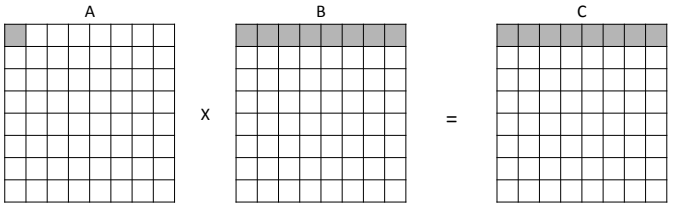
Exemplo: multiplicação de matrizes

- Trocar a ordem dos ciclos para melhorar a localidade espacial no acesso aos dados

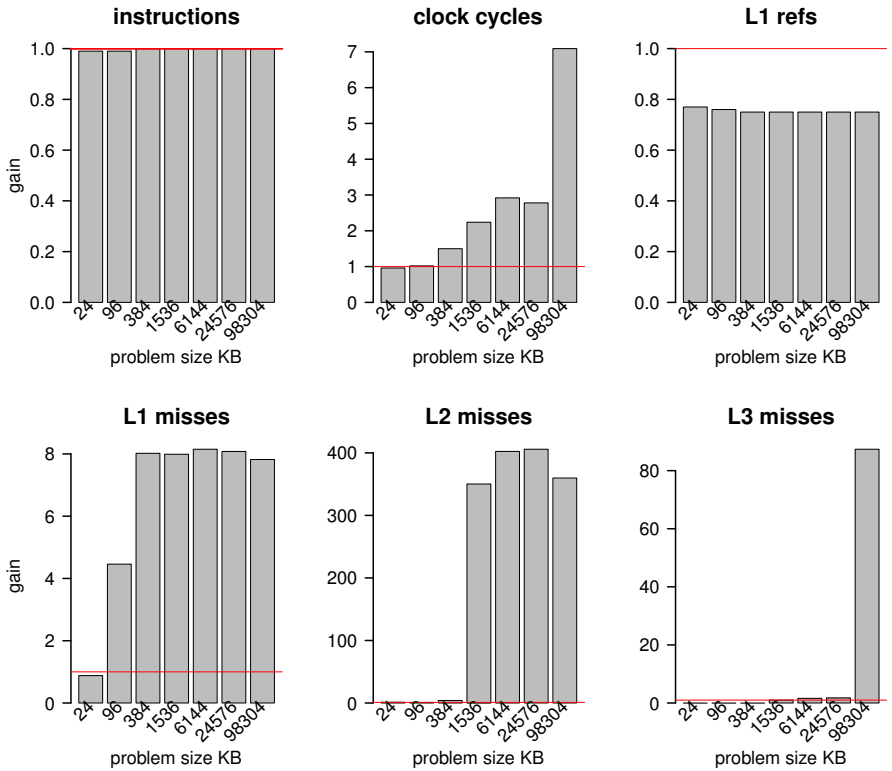
```
for i = 1 to RA
  for j = 1 to CB
    for k = 1 to RB
      C[i][j] += A[i][k]*B[k][j]
```



(a) IJK order



(b) IKJ order



ganho com a passagem da ordem i, j, k para i, k, j

Algoritmos “amigáveis” da hierarquia de memória

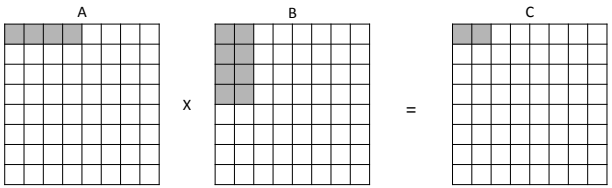
Exemplo: multiplicação de matrizes

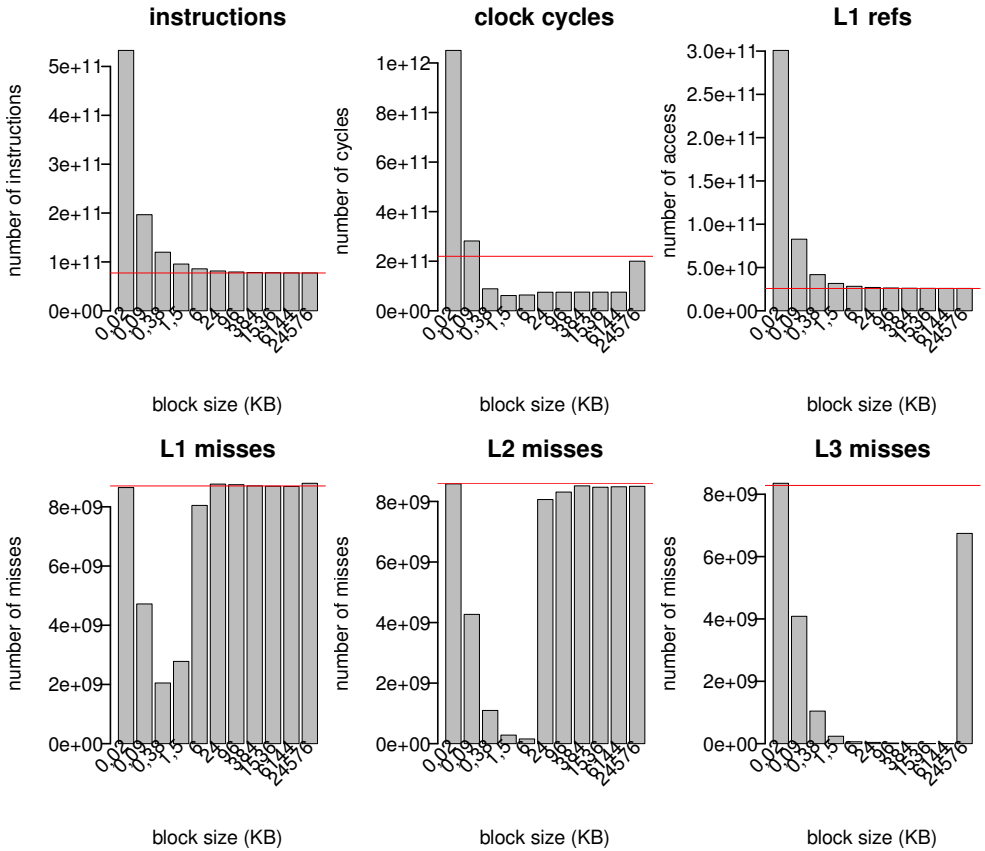
- Aceder aos dados por blocos

```
for i = 1 to RA
   for j = 1 to CB
      for k = 1 to RB
         C[i][j] += A[i][k]*B[k][j]
```

=>

```
for bi = 1 to RA, bi+=blocksize
   for bj = 1 to CB, bj+=blocksize
      for bk = 1 to RB, bk+=blocksize
         for i = bi to bi  + blocksize
            for j = bj to bj + blocksize
               for k = bk to bk + blocksize
                  C[i][j] += A[i][k]*B[k][j]
```



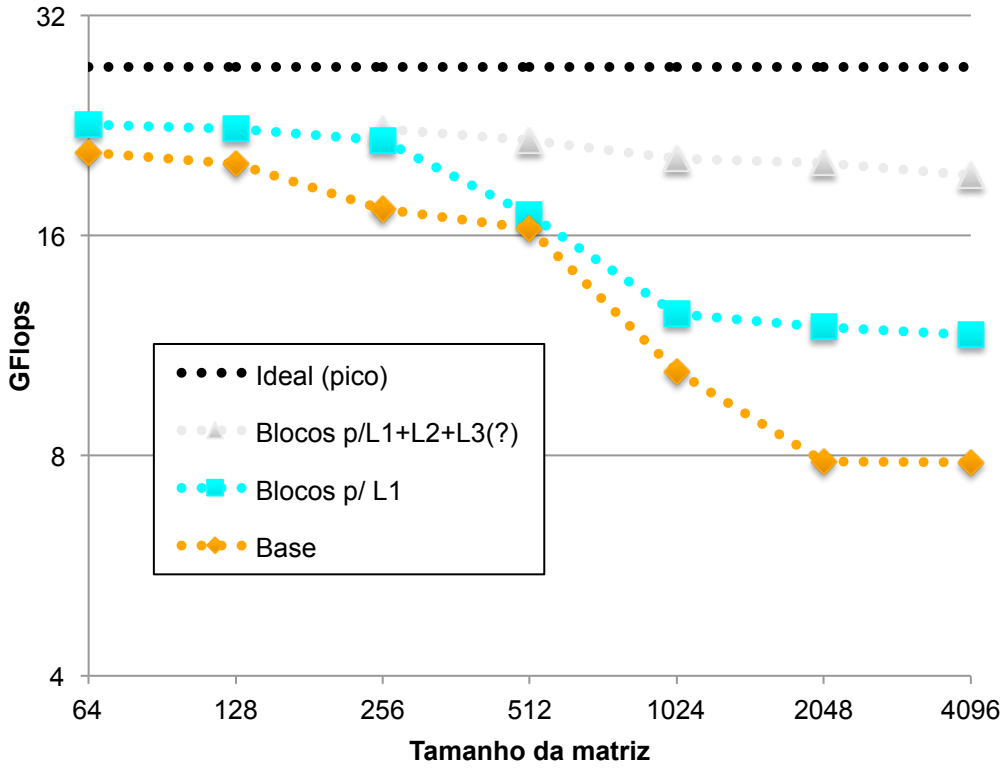


Valores em função do tamanho dos blocos (*blocksize*)

Algoritmos “amigáveis” da hierarquia de memória

Exemplo: multiplicação de matrizes

- Resultados



Tamanho das matrizes	Espaço ocupado (MB)	Notas
64	0,1	Parte cabe na L1
128	0,4	Esgota L1, Cabe na L2
256	1,5	Esgota L2, Cabe na L3
512	6,0	Cabe na L3
1024	24,0	Esgota L3
2048	96,0	
4096	384,0	

Algoritmos “amigáveis” da hierarquia de memória

Exemplo: multiplicação de matrizes

- Caracterização do ambiente experimental:

matrix size of 2048x2048 (∼96MB);

Processor	2∗ (Intel(R) Xeon(R) CPU X5650 2.67GHz 6 HyperTreading)
Cache: - L1 - L2 - L3	 (32KB data + 32KB instruction) ∗ 6 cores (256KB data + instruction) ∗ 6 cores 12288KB data + instruction
Memory	48GB
Operation System	CentOS release 6.2 (Final)
Java	1.7.0_05
C compiler	(GCC) 4.4.6