Variáveis Compartilhadas e Modelos de Memória

Referências

- **Threads Basics**
  Hans-J. Boehm
  http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/threadsintro.html
  Novembro de 2008 / Janeiro de 2011 (revisado)

- **You Don't Know Jack About Shared Variables or Memory Models**
  Hans-J. Boehm e Sarita V. Adve
  Communications of the ACM
  Fevereiro de 2012

Motivação

x++

Motivação

x++

x = x + 1

Motivação

x++

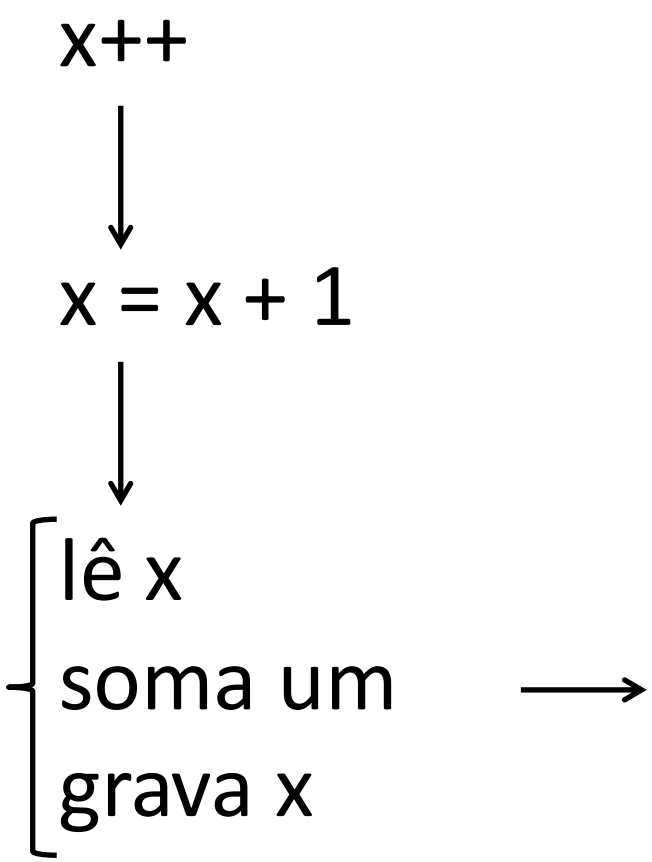
x = x + 1

lê x

soma um

grava x

Motivação



x = 0

lê x	// x = 0
soma um	// x = 0
grava x	// **x = 1**

lê x	// x = 0
soma um	// x = 0
grava x	// **x = 1**

Motivação

x++

↓

x = x + 1

↓

lê x
soma um
grava x

x = 0

lê x // x = 0	lê x // x = 0
soma um // x = 0	soma um // x = 0
grava x // x = 1	
lê x // x = 1	
soma um // x = 1	
grava x // x = 2	
lê x // x = 2	
soma um // x = 2	
grava x // x = 3	
lê x // x = 3	
(...)	
soma um // x = 998	
grava x // **x = 999**	grava x // **x = 1**

não se pode nem pensar em "algumas perdas podem ser toleradas..."

Motivação

- Base decimal;
- Palavras de memória com três dígitos;
- x com seis dígitos.

x++

Motivação

- Base decimal;
- Palavras de memória com três dígitos;
- x com seis dígitos.

x++

tmp_hi = x_hi;

tmp_lo = x_lo;

(tmp_hi, tmp_lo)++;

x_hi = tmp_hi;

x_lo = tmp_lo;

Motivação

- Base decimal;
- Palavras de memória com três dígitos;
- x com seis dígitos.

x++


```
tmp_hi = x_hi;
tmp_lo = x_lo;
(tmp_hi, tmp_lo)++;
x_hi = tmp_hi;
x_lo = tmp_lo;
```

x = 999 (x_hi = 0, x_lo = 999)

```
tmp_hi = x_hi;          // tmp_hi =1
tmp_lo = x_lo;          // tmp_lo=999
(tmp_hi, tmp_lo)++;     // tmp_hi=2
                        // tmp_lo=0
x_hi = tmp_hi;          // x_hi=2
x_lo=tmp_lo;            // x_lo=0
```

```
tmp_hi = x_hi;          // tmp_hi =0
tmp_lo = x_lo;          // tmp_lo=999
(tmp_hi, tmp_lo)++;     // tmp_hi=1
                        // tmp_lo=0
x_hi = tmp_hi;          // x_hi=1
                        // x_lo=999
                        // x=1999

x_lo = tmp_lo;          // x=2000
```

Motivação

- Protocolo de entrada em região crítica por busy waiting.

```
while (!done) {}
... = x;
```

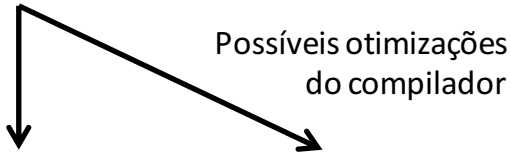
```
x = ...;
done = true;
```

Motivação

- Protocolo de entrada em região crítica por busy waiting.

```
while (!done) {}
... = x;
```

```
x = ...;
done = true;
```

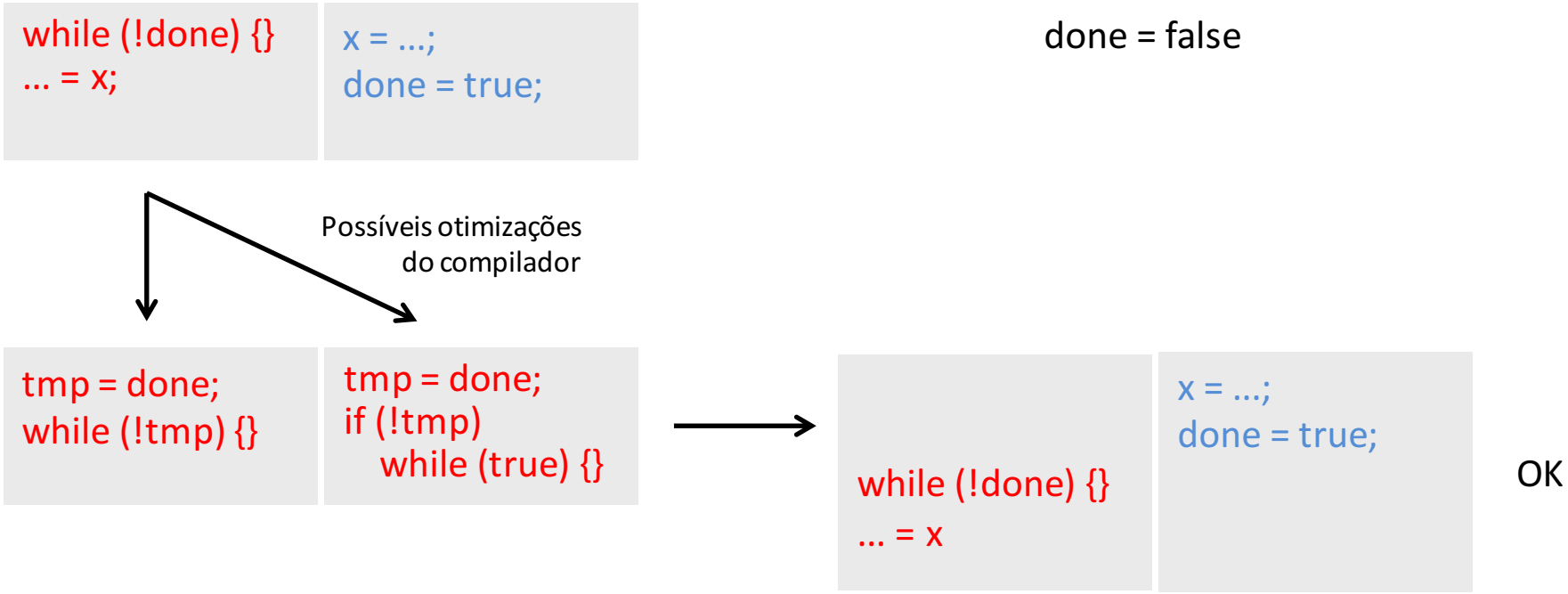



```
tmp = done;
while (!tmp) {}
```

```
tmp = done;
if (!tmp)
   while (true) {}
```

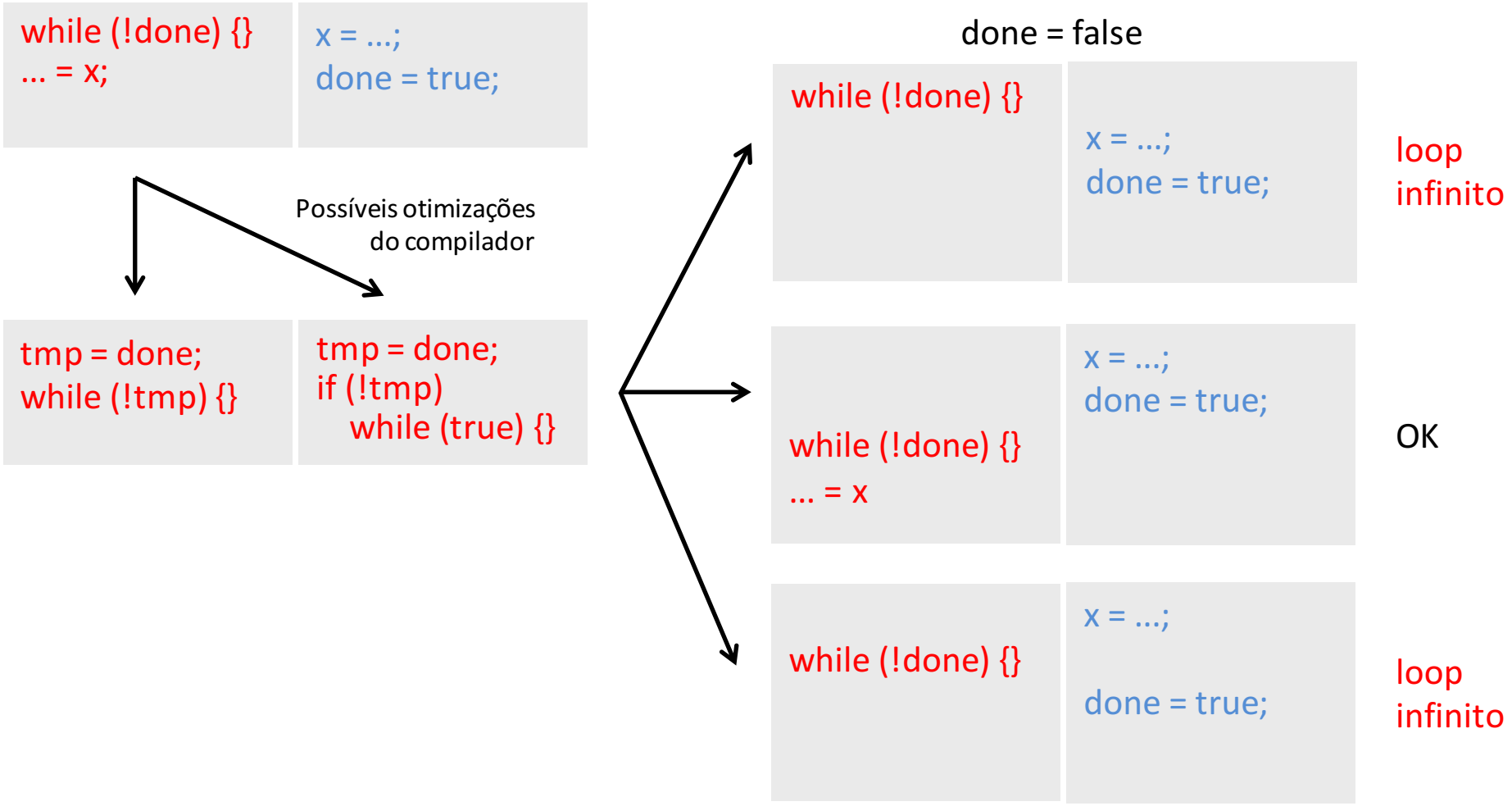
Motivação

- Protocolo de entrada em região crítica por busy waiting.



Motivação

- Protocolo de entrada em região crítica por busy waiting.

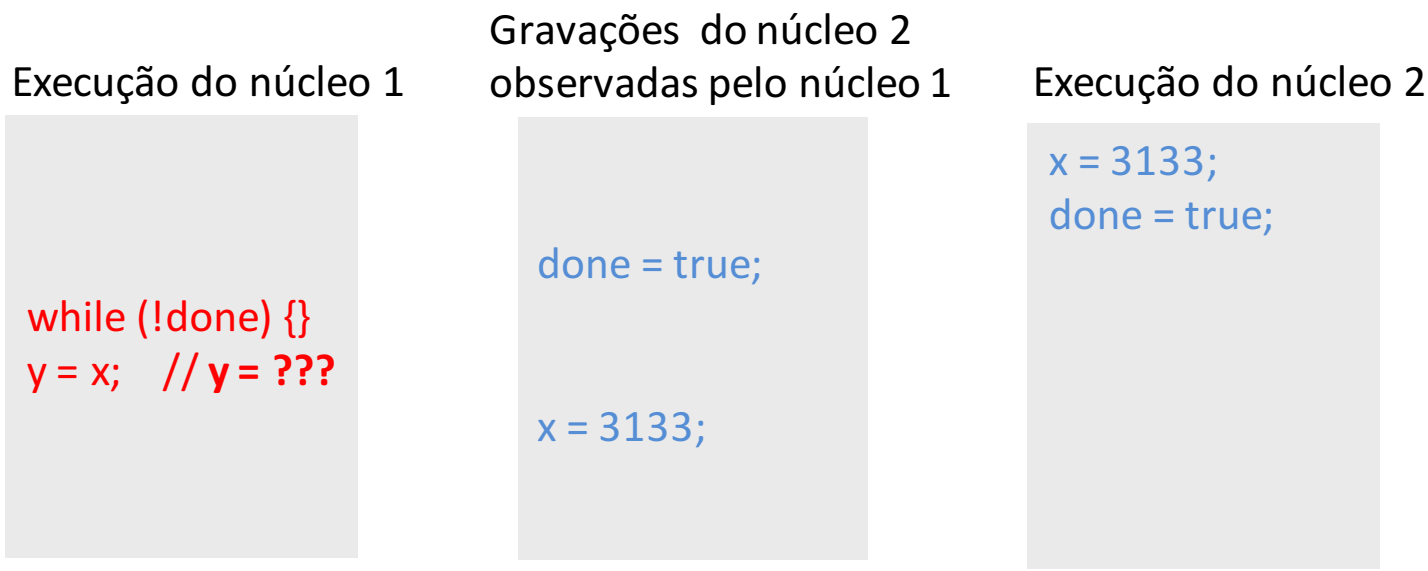


Motivação

- Mesmo na ausência de otimizações do compilador...
- Gravações executadas por um núcleo observadas em ordem distinta por outros núcleos.

Motivação

- Mesmo na ausência de otimizações do compilador...
- Gravações executadas por um núcleo observadas em ordem distinta por outros núcleos.



Conceitos

- Execução de um programa *multithreaded* pode ser vista como execução intercalada de passos de cada *thread*
- execução é "**sequencialmente consistente**" se equivale a alguma execução intercalada desses passos

  – Linguagens como Java, C11 e C++11 oferecem "consistência sequencial" para programas que não possuem condições de corrida.

Conceitos

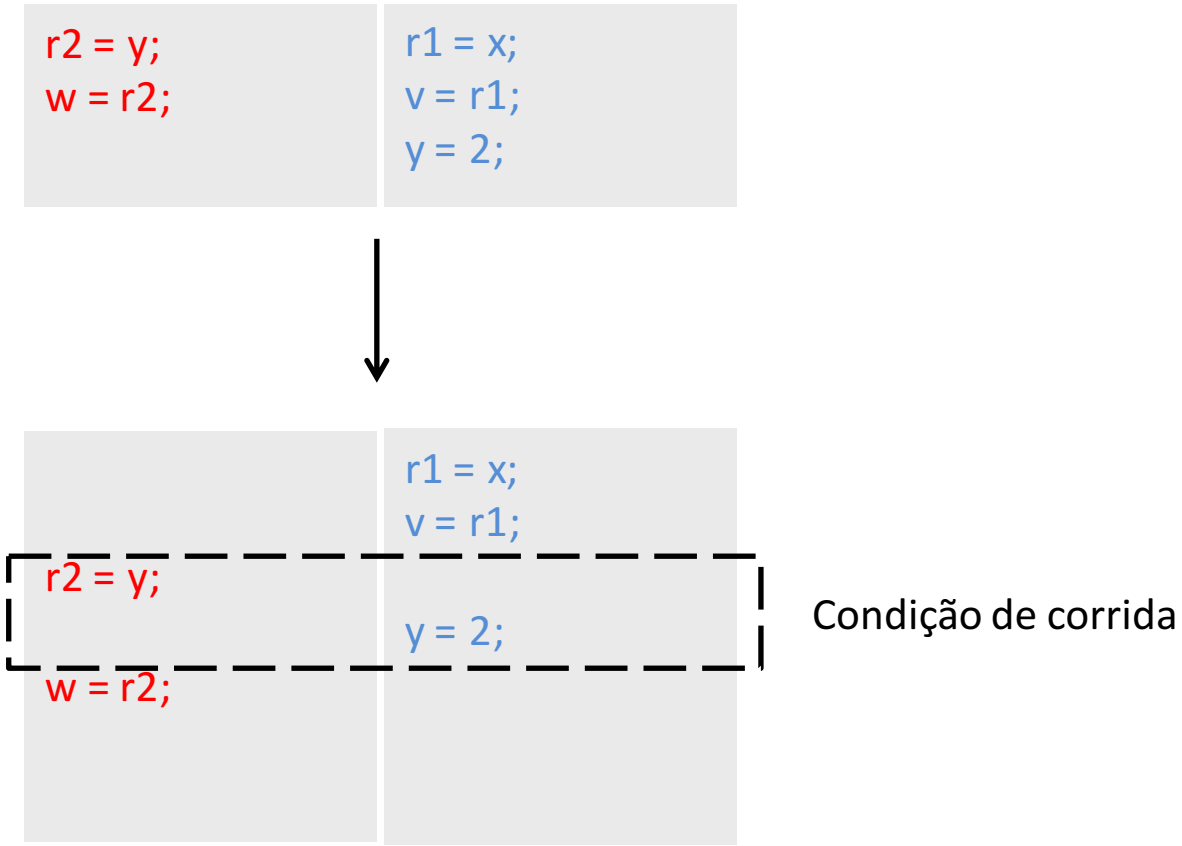
- Duas operações em memória são consideradas conflitantes quando acessam a mesma localização de memória e pelo menos uma delas é uma escrita.
- condição de corriga: operações conflitantes em threads distintas que podem ser executadas “ao mesmo tempo”.
  - execução “ao mesmo tempo” significa executar imediatamente depois na visão intercalada de execução sequencial de passos de cada *thread*.
  - operações de sincronização não contam
- ausência de condições de corrida: não há condições de corrida em nenhuma execução sequencialmente consistente

Conceitos

```
r2 = y;
w = r2;
```

```
r1 = x;
v = r1;
y = 2;
```

Conceitos



Proposta

- Escrever código livre de condições de corrida.
- A implementação garante a “consistência sequencial”.

x = y = false

```
if (x)                    if (y)
   y = true                  x = true
```

Consequências

- Em programas livres de condições de corrida:
  - Percepção de execução atômica de blocos de código sem sincronização;
  - Indiferença em relação à granularidade da atualização da memória (por bytes ou por palavras, por exemplo);
  - Percepção de execução em um único passo de chamadas de bibliotecas sem sincronização interna;
  - Redução da complexidade de raciocinar sobre programas *multithreaded*.

Consequências

- Basta (!) garantir que blocos de código sem sincronização que sejam executados ao mesmo tempo não escrevam ou leiam e escrevam as mesmas variáveis.

- Bibliotecas podem dividir a responsabilidade por evitar condições de corrida entre código do cliente e da biblioteca:
  - Cliente precisa assegurar que duas chamadas simultâneas não acessam o mesmo objeto ou pelo menos não modificam o objeto.
  - Biblioteca precisa assegurar que acessos a objetos distintos e acessos de leitura a objetos não introduzem condições de corrida.

Exclusão Mútua

- Forma mais comum de evitar condições de corrida.
- Não funciona bem com rotinas de tratamento de sinais e interrupções.
- Custo de desempenho.
- Em pthreads, em particular, incidência de bugs e condições de corrida “benignas”.

Variáveis de Sincronização

- Como variáveis normais (de dados), mas acessos são considerados operações de sincronização.
- Admitem acessos a partir de múltiplas *threads* sem implicar condição de corrida.
- Exemplos:
  - volatile int (Java)
  - atomic<int> (C++)
- Adequadas para casos simples de variáveis compartilhadas; inadequadas para estruturas de dados complexas.

Linguagens

- Java:
  - Garante consistência sequencial para programas livres de condições de corrida.
  - Localizações de memória = campos de objeto ou elementos de *array*.
  - Conjunto complexo de regras para definir o comportamento de objetos compartilhados entre threads, inclusive quando ocorrem condições de corrida.
  - Qualquer objeto pode ser utilizado como *lock* para um bloco de código sincronizado:

```
synchronized ( objeto ) {
    região crítica
}
```

  - Suporte também a operações explícitas de *lock*.
  - Variáveis de sincronização (volatile) não podem ser elementos de *array* e garantem atomicidade apenas no acesso à memória.

Linguagens

- C++11:
  - Garante consistência sequencial para programas livres de condições de corrida.
  - Suporte explícito a *threads* na própria linguagem (std::thread).
  - Sem comportamento definido para condições de corrida.
  - Suporte a *locks* com operações explícitas (std::mutex).
  - Variáveis de sincronização (atomic) garantem atomicidade no acesso à memória e em algumas operações (como ++, por exemplo).

estado das coisas

- Garantir ausência de condições de corrida ainda é um problema difícil.
- Avanços recentes:
  - Detecção dinâmica de condições de corrida;
  - Suporte de hardware para geração de exceções para condições de corrida;
  - Anotações de linguagens de programação para eliminar condições de corrida durante a concepção de programas.
- Viabilidade comercial ainda reduzida.