Verificador de Sudoku *Multithread*

Gustavo F. Guimarães, Gustavo Zambonin, Marcello Klingelfus
Sistemas Operacionais I (UFSC-INE5412)

1 Verificação da repetição para um conjunto de números

Funções aludidas: `check_row`, `check_col`, `check_sqr`, `print_errors`

Para evitar a maior complexidade de tempo e espaço obtida a partir de uma solução comum para este problema, uma abordagem mais minimalista foi utilizada, que não envolve desvios condicionais, a criação de vetores auxiliares ou a ordenação do conjunto. É possível perceber que apenas oito bits poderiam ser utilizados para carregar o resultado da verificação da lista de números, pois no caso do *sudoku* sempre existirá um número em sua posição correta. Assim, apenas oito checagens seriam necessárias; entretanto, a prototipação desta estratégia mostrou que a abordagem a seguir teria um melhor balanço entre legibilidade do código e desempenho alcançado.

Sendo assim, o método desenvolvido consiste na alocação de um inteiro *unsigned* de 16 bits, onde apenas nove destes são utilizados. A cada iteração na região selecionada, um deslocamento à esquerda é feito n vezes, onde n é o número presente no vetor na posição referente à iteração. Assim, caso uma região não apresente erros, o inteiro alocado terá valor $(1 << 9) - 1 = 2^9 - 1 = 511$. Em Python, este método pode ser escrito como `reduce(lambda x, y: x | ((1 << y) - 1), range(1, 10))`. Note que é possível substituir o iterável `range` por qualquer lista de números.

```
1  uint8_t s = sqrt(SIZE), r = (sqr_index / s) * s, c = (sqr_index % s) * s;
2  uint16_t repeated = 0;
3  for (uint8_t i = r; i < r + s; ++i) {
4    for (uint8_t j = c; j < c + s; ++j) {
5      repeated |= 1 << grid[i][j];
6    }
7  }
```

Figura 1: Checagem de repetições em uma região quadrada do tabuleiro. Cada uma destas pode ser obtida a partir de um índice $i \in \{0, 8\}$, organizadas da esquerda para a direita, de cima para baixo.

Caso existam repetições, alguns bits não serão ativados, e deste modo, estes podem ser contados para descobrir a quantidade de números repetidos naquela linha. Isto é feito com uma extensão do compilador GCC (também suportada por Clang), chamada de `__builtin_popcount` [1], que conta a quantidade de bits `1` em um inteiro; desse modo, o contador de erros é negado com uma máscara correspondente para que a contagem correta seja feita.

```
1  uint8_t errors = __builtin_popcount(~count & ((1 << SIZE) - 1));
```

Figura 2: Implementação da contagem de erros descrita acima.

{gustavo.g,gustavo.zambonin,marcello.klingelfus}@grad.ufsc.br src/

2 Distribuição das tarefas entre *threads*

Funções aludidas: `main`, `choose_task`, `print_errors`

Solucionar o problema proposto envolve tornar o programa verificador concorrente. Deste modo, é necessário identificar possibilidades de divisão de tarefas a fim de aumentar seu desempenho. Percebe-se que a única tarefa que pode ser dividida é a de verificação de regiões diferentes, visto que existem vinte e sete destas (nove linhas, nove colunas e nove quadrados internos).

O número de *threads* desejadas para a execução do programa é obtido como um argumento fornecido para o programa. É necessário apontar que o número máximo para tal, limitado em $2^{16} - 1$ neste programa, é fundamentado pela seguinte razão:

```
$ man pthread_create | grep "limit" -m4 -A2
       EAGAIN A system-imposed limit on the number of threads was encountered.
              There are a number of limits that may trigger this error: the
              RLIMIT_NPROC soft resource limit (set via setr-limit(2)), which
              limits the number of processes and threads for a real user ID,
              was reached; the kernel's system-wide limit on the number of
              processes and threads, /proc/sys/kernel/threads-max, was reached
              (see proc(5)); or the maximum number of PIDs,
              /proc/sys/kernel/pid_max, was reached (see proc(5)).
$ cat /proc/sys/kernel/threads-max
61085
```

O valor acima foi observado em outros computadores sem maiores variações. Com o número escolhido, um vetor é criado para armazenar as *threads*. Ao inicializá-las, é necessário inserir seus índices como argumento para o método que realiza a distribuição das tarefas.

A distribuição é implementada da seguinte maneira: um vetor de ponteiros de função (declarado com uma extensão da linguagem C que mostra advertências em compilações com configurações mais rigorosas) e um inteiro de 32 bits são criados a fim de representar as tarefas disponíveis. Ao iterar sobre este vetor e escolher uma tarefa, a *thread* mudará o bit referente ao índice desta, assim marcando-a como completa, e executará o ponteiro de função com os argumentos necessários. Este procedimento é sensível à condições de corrida, pois duas *threads* podem escolher a mesma tarefa e mudar o bit duas vezes, de modo que uma terceira poderá escolhê-la novamente. Portanto, mecanismos de exclusão mútua abrangem as partes críticas da função.

```
1   for (uint8_t i = 0; i < length(tasks); ++i) {
2     pthread_mutex_lock(&buf_mutex);
3     if ((available_tasks >> i) & 1) {
4       available_tasks &= ~(1 << i);
5       pthread_mutex_unlock(&buf_mutex);
6       tasks[i](i % SIZE, (intptr_t) tid);
7     } else {
8       pthread_mutex_unlock(&buf_mutex);
9     }
10  }
```

Figura 3: Escolha de tarefas a partir de uma "fila". É possível observar que é necessário liberar a *thread* em dois momentos diferentes pois, do contrário, ela nunca retornará ao fluxo de execução usual, impedindo a continuidade do programa. O ponteiro de função disponível em `tasks` é chamado com dois argumentos: a região escolhida, decidida pelo índice da iteração, que é sempre divisível pelo tamanho do tabuleiro; e o índice da *thread*, convertido de um `void*` proveniente da criação desta.

Outra abordagem possível é uma distribuição prévia das tarefas entre as *threads*; assim, estas já saberiam de suas responsabilidades no momento que são criadas. Essa abordagem alternativa pode prevenir um problema onde uma ou mais *threads* estão sempre tentando obter tarefas mas falhando, assim presas na fila de espera e não oferecendo desempenho máximo.

Após a contagem de erros, caso um exista um elemento repetido, é necessário somar estes ao número total. Como duas *threads* podem modificar este contador em períodos muito próximos, verifica-se que esta região também é crítica; portanto, também é cercada por mecanismos que impedem a contagem errada de erros.

Referências

[1] Free Software Foundation. Other Built-in Functions Provided by GCC, 2017. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>. Acesso em: 22 abr. 2017.