UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

RELATÓRIO
Portabilidade para OpenMP

Alexandre Henrique Soares Dias: *20180001100*

Gabriel Medeiros Coelho: *20180009538*

José Tobias Souza dos Santos: *20180154969*

Natal-RN

2019

1. Introdução

As bibliotecas que utilizam o padrão POSIX para threads, popularmente conhecidas como **pthreads**, são bastante difundidas atualmente, principalmente no ramo da programação paralela em arquiteturas multi-core.

Apesar de seu grande uso e da disponibilidade de ferramentas que essas bibliotecas fornecem, elas ainda consistem em um modelo de programação voltado para baixo nível, com excelência em criação, manipulação e gerenciamento de threads.

Tendo em vista essa limitação, existem outras bibliotecas e interfaces de programação de aplicativo (APIs) disponiveis para arquiteturas com multi-processamento que procurem uma abordagem diferente. Uma delas é o **OpenMP**.

O OpenMP consiste em uma API de alto nível, altamente portável e escalável que pode ser usado no lugar de algumas aplicações que usam pthreads. Uma grande vantagem dessa interface é que, por exemplo, não existe uma limitação de que o código seja escrito em C, como existe em pthreads.

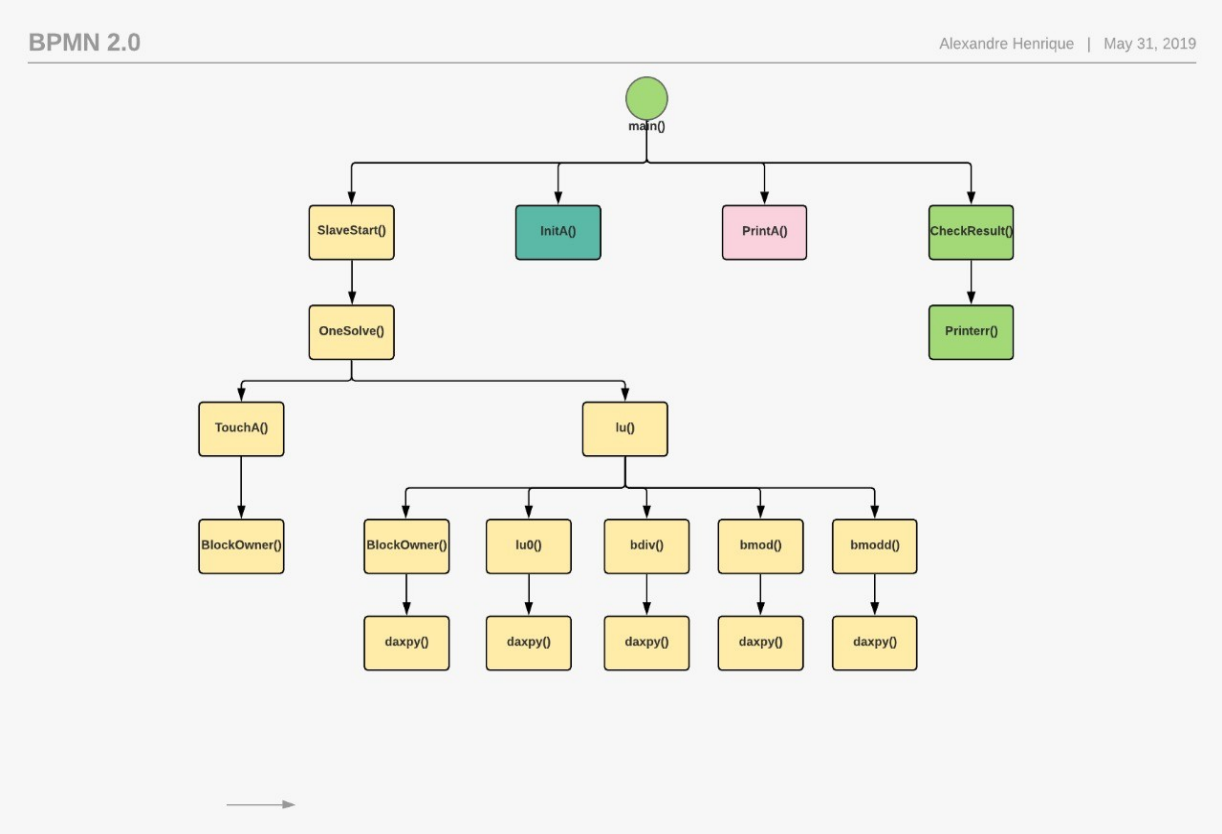
Buscando abstrair um pouco da complexidade de baixo nível apresentada na aplicação LU-CB da suíte *splash2*, bem como aplicar os conceitos obtidos na disciplina, neste relatório serão medidos esforços para portar o código, de pthreads para OpenMP , dessa aplicação.

2. Metodologia

Antes de realizar a portabilidade do código, era necessário desenvolver um entendimento mais profundo do mesmo. Os primeiros esforços tomados foram destinados a compreender a estrutura de chamadas de funções no algoritmo.

2.1 Diagrama de chamadas de função

Analisando o código, pôde-se montar o seguinte diagrama:



E, após analisá-lo, pôde-se ver que:

- Funções como **InitA()**, **PrintA()**, **CheckResult()** e **Printerr()** só necessitam ser executadas por uma thread, pois involvem apenas inicialização, escrita de resultados e escrita de erros.

- Funções responsáveis pela computação em si dos dados são todas chamadas a partir de um ponto em comum: a função **SlaveStart()**.

Logo, é inevitável de se pensar que a portabilidade seja analisada a partir dos pontos em que SlaveStart() é chamada, pois o paralelismo começa a partir dessa função.

2.2 Portabilidade

Analisando o código, dois trechos principais chamaram a atenção. Um deles pode ser visto a seguir:

```
long  i, Error;

for (i = 0; i < (P) - 1; i++) {
  Error = pthread_create(&PThreadTable[i], NULL, (void * (*)(void *))(SlaveStart), NULL);
  if (Error != 0) {
    printf("Error in pthread_create().\n");
    exit(-1);
  }
}

SlaveStart();
```

Esse trecho nada mais faz do que criar as threads e mapear uma execução da SlaveStart() para cada uma. É possível substituí-lo por um simples comando em openMP:

```
# pragma omp parallel num_threads(P)
SlaveStart();
```

A diretiva parallel nada mais é do que o construtor fundamental do openMP, enquanto que a cláusula num_threads(P) especifica o número de threads da região paralela. Logo, após executar esse comando, a thread mestre criará um time de P threads, cada uma responsável por realizar uma parte da computação inteira definida dentro de SlaveStart().

Outra parte do código interessante, é a mostrada a seguir:

```
{pthread_mutex_lock(&(Global->idlock));}
  MyNum = Global->id;
  Global->id ++;
{pthread_mutex_unlock(&(Global->idlock));}
```

Essa parte é executada em várias funções. Aqui, o algoritmo utiliza uma lógica um pouco complexa, com o uso de structs e funções da biblioteca pthreads, apenas para mapear o id, ou nesse caso, myNum, para cada uma das threads. Para fazer isso com a interface do OpenMP, basta realizar uma chamada à função omp_get_thread_num():

```
MyNum = omp_get_thread_num();
```

2.3 Resultados

Após realizada a portabilidade, foi percebido que não haviam outros trechos, dentro das funções que são chamadas por SlaveStart(), que precisassem ser portados. A região paralela foi criada na raiz na chamada das funções, que é justamente a função SlaveStart(), logo, caso fossem criadas outras regiões paralelas dentro das outras funções, haveria uma multiplicação desnecessária do número de threads. Por exemplo, supondo que **m** execuções de SlaveStart() sejam mapeadas para **m** threads e um time de **n** threads seja alocado dentro da função bmod(), essa função contaria agora com **m*n** threads, gerando um trabalho adicional desnecessário. Ao analisar os tempos de execução para tamanhos de problema e números de threads diferentes, antes e depois da portabilidade, pôde-se ver os seguintes resultados:

Tempos de execução do programa (pthread)					
p \ n	512	1024	2048	4096	8192
1	0,195	1,548	12,357	99,494	794,053
2	0,105	0,800	6,289	49,985	398,137
4	0,055	0,414	3,191	25,136	200,095
8	0,036	0,223	1,646	12,871	101,169
16	0,024	0,136	0,879	6,543	51,335
32	0,013	0,068	0,460	3,499	27,897

Tempos de execução do programa (OpenMP)					
p \ n	512	1024	2048	4096	8192
1	0,196	1,553	12,335	99,128	793,343
2	0,102	0,787	6,189	49,284	394,732
4	0,053	0,402	3,114	24,989	197,155
8	0,028	0,207	1,583	12,464	99,046
16	0,015	0,108	0,817	6,324	49,961
32	0,012	0,062	0,436	3,246	25,445

De fato, o resultado é de que, mesmo apenas precisando mudar os trechos de código de SlaveStart(), a portabilidade foi bem sucedida, já que a aplicação continuou sendo escalável. Além disso, com a utilização de OpenMP, várias outras funções e diretivas estão disponíveis para serem utilizadas em outras partes do código. Pode-se, agora, realizar otimizações adicionais, como por exemplo diretivas SIMD que podem efetuar paralelismos de dados (o que antes não era possível com pthreads).

3. Conclusão

Finalmente, uma vez que foi possível portar o código para openMP, um conjunto de novas ferramentas fica disponível para utilização. Isso implica que, além de paralelizar a nível de instrução e de dados, será possível, através do openMP, realizar otimizações que antes não puderam ser observadas, dado o nível de complexidade das funções em pthreads e algumas funções que não pertenciam à regiões paralelas.

Além disso, pôde-se perceber que a interface do openMP é facilmente portável e escalável, dado que um código complexo em baixo nível de pthreads torna-se agora, através de abstrações de alto nivel, muito mais simples.