



Grupo de Estudo de Análise e Técnicas de Sistemas de Potência - GAT

Simulação eficiente de controladores definidos pelo Usuário utilizando compilação em tempo real

Thiago J. Barbosa da Rocha (*)
CEPEL, UFF

Tiago Santana do Amaral
CEPEL

Sergio Gomes Junior
CEPEL, UFF

Leonardo Pinto de Almeida
CEPEL

Luiz Antônio Alves de Oliveira
CEPEL

RESUMO

Este artigo tem como objetivo apresentar uma nova metodologia otimizada desenvolvida para simulação de Controladores Definidos pelo Usuário (CDU). Enquanto essa simulação é realizada pelo ANATEM (Análise de Transitórios Eletromecânicos) a partir da interpretação do arquivo CDU, o novo método proposto consiste na geração de um arquivo compilado em C, resultando em aumento de eficiência computacional. O tempo de simulação é reduzido ainda mais a partir da implementação de processamento em paralelo no momento da execução do código C gerado.

PALAVRAS-CHAVE

CDU compilado, Otimização, processamento paralelo

1.0 - INTRODUÇÃO

O ANATEM (Análise de Transitórios Eletromecânicos) é um programa computacional cujo objetivo é a simulação de transitórios eletromecânicos no domínio do tempo. Desenvolvido pelo Cepel, o ANATEM é importante na operação e no estudo do Sistema Interligado Nacional (SIN). Ele dispõe tanto de controladores de topologia fixa (modelos *built-in*) como também de topologia genérica, possibilitando que o analista faça a modelagem conforme a necessidade de detalhamento do estudo. Para isso, o ANATEM possui blocos a partir dos quais é realizada a modelagem, construindo-se, dessa forma, um Controlador Definido pelo Usuário (CDU).

O objetivo do artigo é propor uma metodologia otimizada para simulação de grande quantidade de controladores, podendo ser utilizada pelo ANATEM e outros programas de simulação, como o ANAHVDC, que realiza a simulação de múltiplos elos de corrente contínua considerando transitórios eletromecânicos e eletromagnéticos. Tal otimização é de grande importância uma vez que uma parte do tempo computacional considerável exigida pelo ANATEM se refere à solução dos CDUs.

No estágio atual, o ANATEM atua por interpretação dos dados de um CDU, da mesma forma que linguagens como o Matlab (1). Linguagens que atuam por interpretação realizam a execução diretamente do código fonte, em uma única etapa. Por outro lado, em linguagens compiladas são necessárias duas etapas: primeiro há a conversão do código fonte em código de máquina (compilação) e em seguida a execução desse código (2).

Uma das vantagens da abordagem interpretada é que ela independe do sistema operacional. Além disso, no método compilado é necessária a geração do executável ou biblioteca de vínculo dinâmico (DLL) a cada vez que o CDU é alterado, o que não acontece no ANATEM, uma vez que ele atua por interpretação (2). Porém, se o CDU não for modificado, há apenas a etapa de execução no método compilado. Com isso, diversos eventos podem ser simulados sem que seja necessário alterar o código binário gerado, o que faz com que seja eliminada a fase de compilação. Nesse caso, o fator primordial é o tempo de execução (*runtime*) e, nesse caso, a implementação

(*) Avenida Horácio Macedo, n° 354 – Cidade Universitária – CEP 21.941-911 Rio de Janeiro, RJ – Brasil Tel: (+55 21) 2598-6131 – Fax: (+55 21) 2598-6131 – Email: thiagojose@id.uff.br

compilada apresenta vantagem. Isso acontece porque o ANATEM precisa varrer o CDU toda vez que ele é usado, mesmo que nenhuma modificação seja realizada. Considerando-se o caso do SIN, que é alterado apenas semestralmente e possui mais de cem mil linhas, o impacto de realizar a interpretação para cada simulação de evento é significativo e pode ser eliminada com a abordagem do CDU compilado.

Portanto, diante da necessidade de aumento de eficiência computacional com redução do tempo de execução, foi implementado um método em que o CDU fosse compilado ao invés de interpretado pelo ANATEM. O resultado foi um programa escrito em C++, que é o tema do artigo. O programa realiza a leitura de um CDU e gera um código em C, que é compilado em tempo real e embutido no próprio programa, por exemplo, o ANATEM. Durante a simulação é possível empregar processamento paralelo, que contribui ainda mais para uma maior eficiência computacional.

Acredita-se que as contribuições técnicas deste artigo irão impactar positivamente na questão da eficiência computacional das simulações do SIN, permitindo a simulação de sistemas com maiores dimensões, com maior grau de detalhamento e com uma maior quantidade de casos, contribuindo para a melhoria das análises dinâmicas que são realizadas na atualidade.

2.0 - PROGRAMAÇÃO COM CONCORRÊNCIA

As simulações foram realizadas em dois computadores. O primeiro, com melhor desempenho, foi um Intel i9-9900k @ 3,60GHz (5 GHz com Intel Turbo Boost), com 8 núcleos (16 núcleos lógicos). O outro foi um Intel i7-6700HQ @ 2,60 GHz (3,50 GHz com Intel Turbo Boost) e com 4 núcleos (8 núcleos lógicos).

Quanto à concorrência é possível uma abordagem que envolva múltiplas *threads* para um mesmo processo. Nesse caso, duas ou mais atividades estão em desenvolvimento ao mesmo tempo, porém não simultaneamente¹. A ilusão da simultaneidade é obtida a partir da alternância rápida entre as threads (*task switching*) (3). Outra possibilidade, e que exige uma máquina *multi-core*, é o uso de apenas uma *thread* por núcleo e, assim, cada processo é enviado a um núcleo diferente (ainda que não seja obrigatório que cada núcleo esteja associado a um processo do programa). Dessa forma, as *threads* estão obrigatoriamente sendo executadas simultaneamente, o que é conhecido como *hardware concurrency* ou paralelismo (3). Geralmente o que acontece é uma combinação de ambas as abordagens, sendo cada núcleo responsável por múltiplas *threads*.

3.0 - PROGRAMA

Didaticamente, o programa pode ser dividido em duas etapas. A primeira consiste na geração de uma DLL com as informações contidas no arquivo CDU lido. A outra etapa é a simulação propriamente dita.

Inicialmente, na primeira etapa, o programa realiza a leitura linha a linha do arquivo CDU, de modo a importar os seus dados. Cada arquivo pode conter diversas unidades de CDU, cada qual identificada por um nome e um número. Dessa forma, cada unidade deve ser tratada separadamente. A partir dessas informações o programa gera um código em C, contendo os dados de cada bloco e a associação com seus respectivos parâmetros fixos e variáveis. A escolha da linguagem do código gerado ainda será revisada para a versão final do programa, podendo ser em C, em Assembly ou em qualquer outra linguagem que seja considerada mais conveniente. O último passo dessa etapa é a geração de uma DLL a partir da compilação do código anterior com uso de um compilador embutido no próprio programa, que também será decidido para a versão final. No artigo foram usados como compiladores o Clang, o GNU Compiler Collection (GCC) e o Tiny C Compiler (TCC).

Na segunda etapa, a DLL gerada é usada para as simulações dos eventos. Dessa forma, o usuário pode escolher dentre diversas opções conforme a FIGURA 1. Como mencionado, é possível que o usuário insira o número de threads desejada (instrução -c). Além disso, como cada DCU pode conter diversos CDUs, é possível a simulação de um único CDU ou de múltiplos CDUs. Neste último caso, eles podem ser inseridos individualmente ou a partir de um intervalo (instrução -m).

Dentre os 85 blocos disponíveis no ANATEM buscou-se focar nos blocos de maior ocorrência (5). Ao final, foram implementados e validados 57 blocos no CDU compilado. Com isso, já seria possível simular 877 CDUs do SIN, o que corresponde a 75% do total. A inicialização e posterior solução de cada bloco foi validada para três casos:

- 1) Aplicação de degrau no primeiro bloco (ENTRAD) e valor de inicialização (DEFVAL) fornecido para o mesmo bloco
- 2) Sem aplicação de degrau e valor de inicialização fornecido para o primeiro bloco (ENTRAD)

¹A Intel implementou a tecnologia de *Hyper-threading*, que consiste em dois processadores lógicos para cada núcleo físico. Cada núcleo lógico pode, geralmente, ser individualmente interrompido e ordenado a executar determinada thread independente do outro. Entretanto, a unidade aritmética, por exemplo, deve ser compartilhada entre os processadores lógicos, ou seja, duas *threads* não podem fazer uso simultâneo da unidade. Desse modo, não é obtida um paralelismo pleno, o que é demonstrado pelo fato do ganho do uso de *Hyper-threading* ser de até 30% e não de 100%, o que seria esperado caso houvesse dois núcleos físicos(4).

3) Sem aplicação de degrau e valor de inicialização fornecido para o último (SAIDA).

- a Seta a amplitude do degrau
- c Seta o número de threads concorrentes a serem utilizadas
- d Seta o intervalo de tempo da simulacao (delta T)
- e Seta o tempo em que o evento deve ser aplicado
- f Seta o tempo final
- g Seta a granularidade da distribuição dos CDUs entre as threads
- h Mostra essa mensagem de ajuda
- i Seta o tempo inicial
- m Sinaliza insercao de CDUs por intervalo
- n Indica que não deve contabilizar nenhuma variavel para a plotagem (nem as saídas)
- o Seta o nome do arquivo de plotagem
- p Indica que deve plotar todas as variáveis (e não apenas as saídas)
- v Seta o indice da variavel em que o degrau sera aplicado

FIGURA 1 – Opções de Simulação

Dessa forma, nos dois primeiros casos a inicialização acontece da entrada para a saída, enquanto no terceiro caso se dá o contrário. Em alguns blocos específicos, como blocos lógicos, a inicialização só é possível em um dos sentidos e, por isso, não foi possível simular os três casos. Na FIGURA 2 é apresentado o diagrama de blocos para o CDU usado para o caso teste do bloco Print. Por sua vez, a saída (variável X3) da simulação do primeiro caso (com aplicação de degrau) do bloco Print está retratada n

FIGURA 3. É possível ver na figura a dinâmica proveniente do bloco. O tempo de simulação dos casos teste foi de 10 segundos e o passo foi de 10^{-5} segundo.

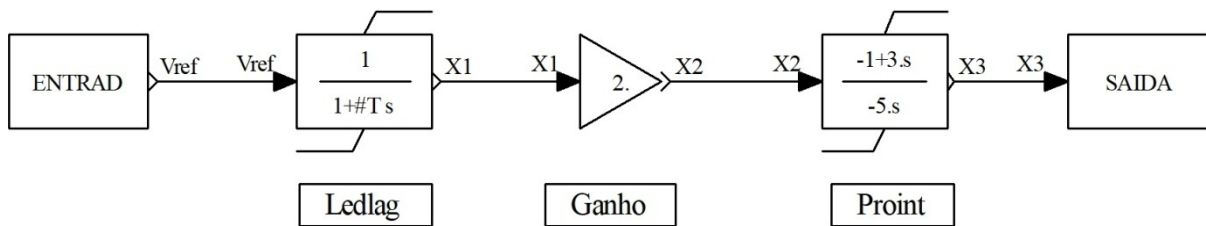


FIGURA 2 – Diagrama de Blocos (Caso teste do bloco Print)

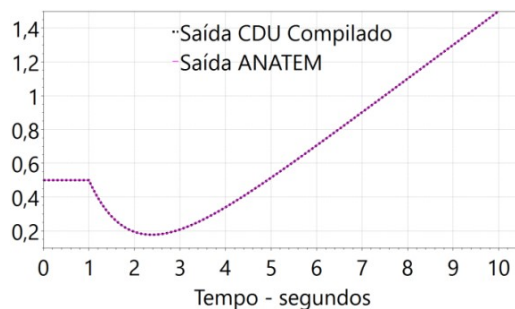


FIGURA 3 – Saída do Caso Teste do Bloco Print

Os blocos IMPORT e EXPORT ainda não foram implementados no CDU compilado. Por isso, foram substituídos, por blocos ENTRAD e SAIDA, respectivamente, e o valor setado a partir de um DEFVAL.

4.0 - METODOLOGIA DE COMPILAÇÃO DOS CONTROLADORES

Neste item será mostrada a metodologia de compilação dos CDUs a partir do seguinte sistema exemplo:

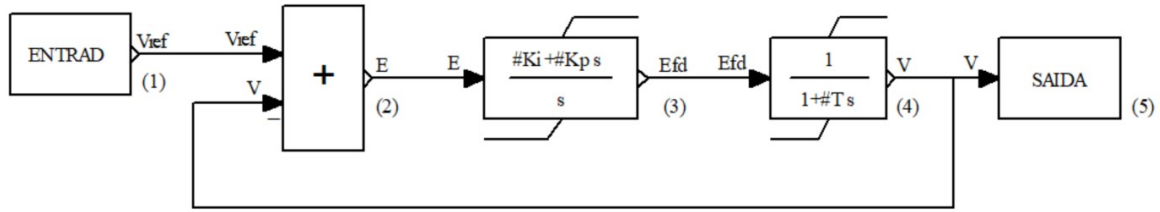


Figura 4– Controlador Exemplo

Segue um código C que soluciona este CDU:

```
void solcdu(double *b01, double *c01, double *x)
{
  x[1]=x[0]-x[3];          //e=Vref-V;
  x[2]=c01[0]*x[1]+b01[0]; //Efd=c01[0]*e+b0[0];
  x[3]=c01[1]*x[2]+b01[1]; //V=c01[1]*Efd+b0[1];
}
```

FIGURA 5–Código C para solução do controlador exemplo

Neste CDU há a presença de blocos dinâmicos que são resolvidos pelo método trapezoidal de solução. A título de ilustração será apresentada a aplicação do método trapezoidal ao Bloco #4 que é um atraso simples de primeira ordem que exige o cálculo de constantes e termo histórico. Segue a equação no domínio s do bloco:

$$Y(s) = \frac{1}{1 + sT} U(s) \Rightarrow (1 + sT) Y(s) = U(s) \quad (1)$$

Pode-se então transformar para o domínio do tempo pela substituição da variável de Laplace s pelo operador derivada:

$$T \frac{dy(t)}{dt} + y(t) = u(t) \quad (2)$$

Aplica-se então a regra trapezoidal em que a equação diferencial é substituída por uma equação a diferenças das variáveis no tempo atual e anterior, assumindo uma variação linear no tempo e a aplicação ao ponto médio:

$$T \frac{y(t) - y(t - \Delta t)}{\Delta t} + \frac{y(t) + y(t - \Delta t)}{2} = \frac{u(t) + u(t - \Delta t)}{2} \quad (3)$$

Então, simplifica-se a equação para explicitar o valor da saída em função da entrada e dos valores das variáveis no passo anterior:

$$\begin{aligned} \left(\frac{2T}{\Delta t} + 1\right) y(t) &= u(t) + u(t - \Delta t) + \left(\frac{2T}{\Delta t} - 1\right) y(t - \Delta t) \\ &= \frac{1}{\left(\frac{2T}{\Delta t} + 1\right)} [u(t) + u(t - \Delta t)] + \frac{\left(\frac{2T}{\Delta t} - 1\right)}{\left(\frac{2T}{\Delta t} - 1\right)} y(t - \Delta t) \end{aligned} \quad (4)$$

$$y(t) = c01 u(t) + b01(t - \Delta t) \quad (5)$$

Em que:

$$c01 = \frac{1}{\left(\frac{2T}{\Delta t} + 1\right)}, c02 = \frac{\left(\frac{2T}{\Delta t} - 1\right)}{\left(\frac{2T}{\Delta t} - 1\right)}, b01(t) = c01 u(t) + c02 y(t) \quad (6)$$

Voltando-se ao código exemplo, verifica-se que há argumentos da função de solução no C correspondentes às constantes e aos termos históricos. A seguir são apresentadas as funções que realizam este cálculo:

```

void ctecdu(double *c01, double *c02, double dt)
{
#define par_Ki 50
#define par_Kp 0.1
#define par_T 1.0
c01[0] = par_Ki*dt/2 + par_Kp;
c02[0] = par_Ki*dt/2 - par_Kp;
c01[1] = 1/(2*par_T/dt+1);
c02[1] = (2*par_T/dt-1)*c01[1];
}

```

FIGURA 6 – Código C para cálculo das constantes do controlador exemplo

```

void memcdu(double *b01, double *c01, double *c02, double* x0)
{
b01[0] = c02[0]*x0[1]+x0[2];
b01[1] = c02[1]*x0[3]+c01[1]*x0[2];
}

```

FIGURA 7 – Código C para cálculo dos termos históricos

Na metodologia apresentada, o programa processa os CDUs e gera o código fonte análogo ao apresentado na FIGURA 5, na FIGURA 6 e na FIGURA 7 e gera uma DLL que pode ser chamada pelo algoritmo de solução propriamente dito e que pode incluir a solução das equações da rede e modelos de equipamentos como máquinas síncronas, elos HVDC e equipamentos FACTS. A título de ilustração é apresentado na FIGURA 8 um código principal que calcula apenas o controlador e guarda em um vetor.

```

voidsolveCDU() {
int i, n;
double u, u0, *y, x[4], x0[4], t, dt, tmax, tdist, b01[2], c01[2], c02[2],
erro, yold;
const double tol = 1e-6;
t = 0;
dt = 0.02;
tmax = 10;
tdist = 1;
u = 0;
n = int(tmax / dt + 1);
y = new double[n];
// Inicializações
t = 0.0;
y[0] = 0.0;
inicdu(x, x0);
// Cálculo das constantes utilizadas na integração numérica de blocos
// dinâmicos
ctecdu(c01, c02, dt);
// Loop do tempo
int it = 0;
while (t <= tmax) {
t = t + dt;
it++;
for (i = 0; i < 4; i++) x0[i] = x[i];
u0 = u;
if (t > tdist + dt / 2) u = 1;
x[0] = u;
memcdu(b01, c01, c02, x0);
// Loop de solucao
erro = 10 * tol;
while (erro > tol) {
yold = x[3];
solcdu(b01, c01, x);
erro = (x[3] - yold) * (x[3] - yold);
}
y[it] = x[3];
}
}
}

```

FIGURA 8 – Código C para solução do CDU exemplo

Ao todo, foram simulados e validados 285 CDUs do SIN. Na FIGURA 9 é apresentada uma versão adaptada do regulador de Tensão de Itumbiara. Originalmente os efeitos da rede são transmitidos ao regulador de velocidade por meio do bloco IMPORT, enquanto o resultado provindo do regulador é transmitido à rede por meio do bloco EXPORT. Como ambos os blocos não foram implementados no programa, o efeito desses blocos foi simulado pela introdução de uma função de transferência, criando uma malha adicional de retroalimentação que emula a relação entre a tensão terminal e a tensão terminal de terceira ordem que considerando o modo da excitatriz e um par complexo conjugado do modo eletromecânico. Além disso, foi inserido um bloco soma, permitindo que seja aplicado um degrau no sistema. Na FIGURA 10 são mostrados os blocos que compõem a função de transferência do regulador de tensão, além do bloco soma.

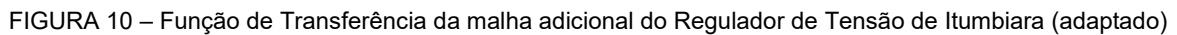
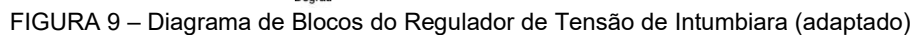


FIGURA11 – Saída da Simulação do Regulador de Tensão de Itumbiara (adaptado)

Em uma segunda simulação foi usado como base o Regulador de Velocidade de Angra 1. Novamente, os blocos IMPORT e EXPORT foram removidos e foi criada uma malha adicional de retroalimentação que relaciona a velocidade da máquina e sua potência mecânica, emulando a equação de oscilação de um gerador. Nesse caso a função de transferência se resumiu ao bloco Print. Além disso, foi criado um somador para a aplicação do degrau, conforme FIGURA 12.

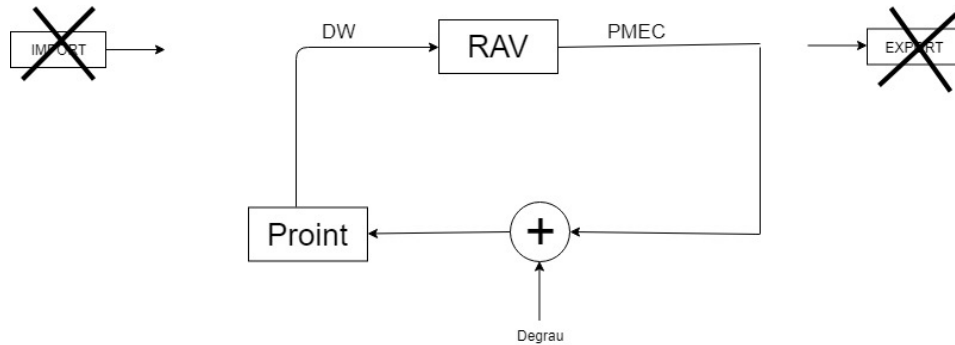


FIGURA 12 – Diagrama de Blocos do Regulador de Velocidade de Angra 1(adaptado)

A saída do CDU se encontra na FIGURA 13. Da mesma forma que a primeira simulação, foi aplicado um degrau de amplitude 0,1 no instante 1 segundo. Nesse caso, a potência também sofreu um afundamento e o sistema encontrou um novo ponto operação, não perdendo a estabilidade.

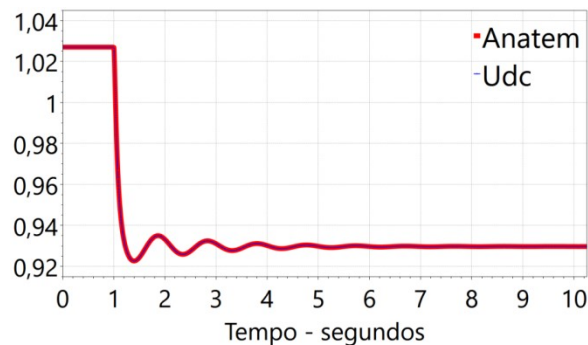


FIGURA 13 – Saída da Simulação do Regulador de Tensão de Itumbiara (adaptado)

Na FIGURA14 são apresentados os tempos de solução de todos os CDUs que compõe o SIN no computador Intel i9. Cada cor de coluna corresponde a um compilador. No primeiro grupo de colunas do eixo horizontal é apresentado o tempo de solução para o caso sem concorrência, ou seja, nenhuma *thread* é criada no decurso do programa exceto a já lançada automaticamente no seu início. Esse caso é indicado por zero no eixo horizontal. Os demais valores no eixo horizontal se referem às *threads* lançadas desconsiderando-se aquela lançada automaticamente. De modo geral, o aumento de threads gera uma redução do tempo de solução até um ponto ótimo. A partir desse ponto o aumento no número de *threads* resulta em piora de desempenho. Isso acontece porque a criação de *threads* é uma etapa de alto custo computacional e, dessa forma, esse aumento de tempo deve ser compensado pelo ganho proporcionado pela *thread*. Dessa forma, o número ótimo de threads depende do computador utilizado e dos processos concomitantes que estão acontecendo no momento de execução da solução. Por isso, o programa possibilita que o usuário selecione o número desejado de *threads*, sendo 4 o valor *default*.

Para o Intel i9 é possível perceber na FIGURA14 que 16 threads foi o número ótimo para todos os compiladores. Como esperado, o Clang e o GCC foram os compiladores que obtiveram o melhor desempenho, ambos utilizando como otimização o padrão -ofast. Para o caso ótimo com 16 threads a solução, tanto pelo Clang como pelo GCC, foi cerca de 11 vezes mais rápida do que pelo ANATEM. Por sua vez, o ganho no TCC foi da ordem de 7 vezes.

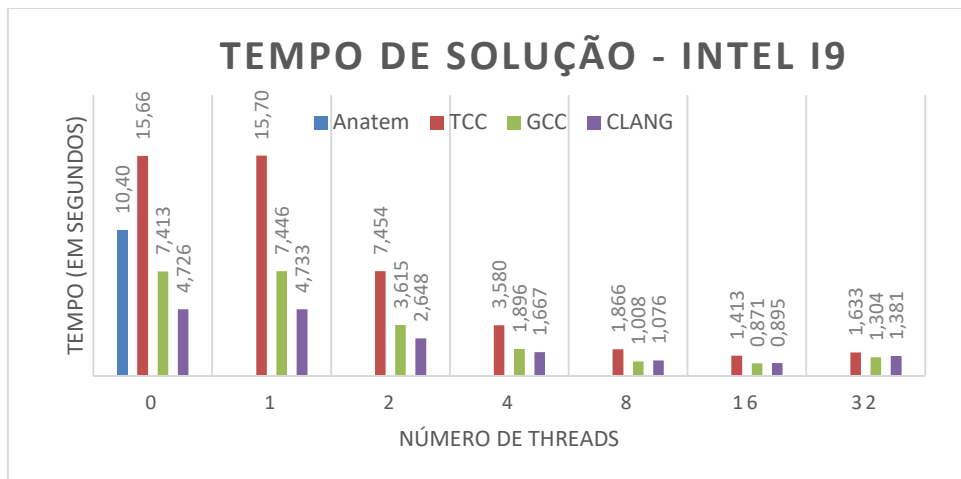


FIGURA14 – Tempos de Solução (Intel i9)

Por sua vez, o desempenho do Intel i7 é retratado na FIGURA 15. Dessa vez, o número ótimo de threads dependeu do compilador utilizado: 8 threads para o GCC e Clang e 16 threads para o TCC. Novamente, GCC e Clang obtiveram os melhores resultados. Para o caso ótimo, em relação ao ANATEM, o GCC foi 6,3 vezes mais rápido e o Clang 7,3 vezes. O TCC teve um ganho de cerca de 3,5 vezes.

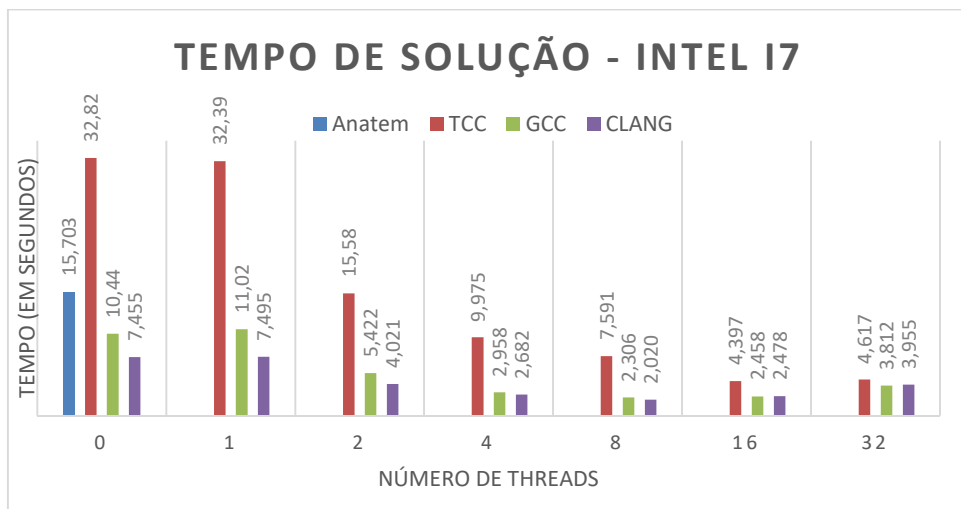


FIGURA 15 – Tempos de Solução (Intel i7)

6.0 - CONCLUSÃO

Os resultados obtidos revelam que os objetivos propostos foram atingidos. As simulações realizadas com o CDU compilado foram validadas pelo ANATEM para diversos casos e usando-se todos os blocos já implementados. A partir de compiladores altamente otimizados, como o Clang e o GCC, foi possível a obtenção de redução de tempo em relação ao ANATEM da ordem de até 11 vezes. Mesmo no caso do TCC, que gera um código com menos otimizações, ainda foi obtida uma redução de tempo de até 7 vezes. A abordagem por CDU compilado é de grande importância na simulação de CDUs com número elevado de blocos. Espera-se que o novo método possa ser aplicado no próprio ANATEM e no futuro ANAHVDC.

7.0 - REFERÊNCIAS BIBLIOGRÁFICAS

- (1) CHAPMAN, S., "MATLAB Programming for Engineers", 6ª Edição, Editora Cengage Learning, USA, 2019.
- (2) TUCKER, A., "Computer Science Handbook", 2ª edição, Editora Chapman and Hall/CRC, USA, 2004.
- (3) WILLIAMS, A., "C++ Concurrency in Action: Practical Multithreading", 2ª Edição, Editora Manning Publications, USA, 2019.
- (4) INTEL® Hyper-threading Technology. Disponível em: <http://www.cslab.ece.ntua.gr/courses/advcomparch/2007/material/readings/Intel%20Hyper-Threading%20Technology.pdf>
- (5) MANUAL DO ANATEM. Brasil.

8.0 - DADOS BIOGRÁFICOS

	Thiago José Barbosa da Rocha graduou-se em Engenharia Elétrica em 2015 pela Universidade Federal Fluminense (UFF). Atualmente é mestrando em Engenharia Elétrica e de Telecomunicações pela Universidade Federal Fluminense (UFF) e bolsista do Centro de Pesquisa de Energia Elétrica (CEPEL). Sua principal área de interesse é a simulação de transitórios eletromagnéticos.
	Tiago Santana do Amaral graduou-se em Engenharia Elétrica em 2004 pela Universidade Federal do Rio de Janeiro, concluiu Mestrado também em Sistemas de Potência pela COPPE/UFRJ em 2007. Atualmente é doutorando em sistemas de potência pela COPPE/UFRJ. Desde 2006 é pesquisador do Cepel trabalhando na pesquisa e desenvolvimento de programas computacionais para a análise de sistemas de potência. Suas principais áreas de interesse são: confiabilidade, controle e modelagem computacional de sistemas de potência
	Sergio Gomes Junior graduou-se em Engenharia Elétrica em 1992 pela Universidade Federal Fluminense, concluiu Mestrado e Doutorado também em Engenharia Elétrica pela Universidade Federal do Rio de Janeiro em 1995 e 2002, em 2004 fez um pós-doutorado na Northeastern University em Boston, Estados Unidos e em 2016 um pós-doutorado na Norwegian University of Science and Technology em Trondheim, Noruega. Desde 1994 é pesquisador do Cepel trabalhando na pesquisa e desenvolvimento de programas computacionais para a análise de sistemas de potência e desde 2000 é gerente do projeto PacDyn no Cepel. Desde 2010 também é professor da Universidade Federal Fluminense. Suas principais áreas de interesse são: dinâmica e controle de sistemas de potência, eletrônica de potência, harmônicos e transitórios eletromagnéticos. É Senior Member do IEEE e membro do Comitê de Estudos B4 do Cigré-Brasil.
	Leonardo Pinto de Almeida graduou-se em Engenharia Elétrica pela UFRJ em 2000, obteve o título de Mestre na área de Sistemas de Potência na COPPE/UFRJ em 2004 e atualmente está cursando Doutorado na UFF. Desde 2000 é pesquisador do CEPEL, onde atua no Departamento de Redes Elétricas (DRE). Trabalha principalmente na área de estudos elétricos para o planejamento da operação e expansão do sistema elétrico brasileiro. É membro do Comitê de Estudos B4 (CCAT e Eletrônica de Potência) do Cigré-Brasil. Suas principais áreas de interesse são: dinâmica e controle de sistemas de potência, eletrônica de potência, transitórios eletromagnéticos, transmissão em corrente contínua (CCAT) e sistemas com múltiplas alimentações CCAT.
	Luiz Antonio Alves de Oliveira graduou-se em Matemática pela UFRJ em 1998, graduado em Informática pela UERJ em 2001, M.Sc. em Informática-Métodos Numéricos e Otimização pelo IM-UFRJ em 2002 e D.Sc. em Engenharia de Sistemas e Computação-Otimização pela Coppe-UFRJ em 2013. Desde 2006 trabalha no CEPEL como pesquisador do Departamento de Redes Elétricas-DRE, tendo trabalhado no projeto MODPOL (Modernização dos Programas Computacionais de Planejamento e Operação Elétricos). Recentemente trabalha também no projeto Siger (Sistema de Gestão de Dados de Redes Elétricas).