Paralelização do Problema do Despacho Econômico Utilizando Evolução Diferencial com OpenMP

Relatório Técnico

Sandra Mara Guse Scos Venske[1]

[1]Universidade Federal do Paraná (UFPR)

sandram@inf.ufpr.br

1. Introdução

Computadores multicores estão por toda a parte. Projetos multicores vêm se tornando dominantes para o futuro de microprocessadores de alta performance e seu uso permite melhorias contínuas significativas de desempenho para aplicações com alto grau de paralelismo [Humenay et al. 2007]. Este fato deve impulsionar programadores a adquirirem experiência no projeto e desenvolvimento de aplicativos que explorem técnicas de programação paralela. Seguindo o mesmo raciocínio, algumas linguagens, como C e Fortran, têm se expandido a fim de oferecer suporte ao paralelismo. OpenMP e MPI são exemplos de tais expansões [Quinn 2003].

A Evolução Diferencial faz parte do grupo de algoritmos evolutivos, como algoritmos genéticos e programação genética. Este grupo pertence a um grupo maior, conhecido como Computação Natural, que ainda engloba outros algoritmos de Inteligência Computacional como Redes Neurais e Sistemas *Fuzzy*.

A exploração do paralelismo inerente às técnicas da Computação Natural, geralmente, possibilita o desenvolvimento de algoritmos mais plausíveis biologicamente, ou seja, que melhor refletem o que realmente ocorre na natureza. Além disso, a implementação paralela das técnicas de computação natural ainda faz com que as mesmas se tornem mais robustas (tolerantes a falhas) e eficientes (mais rápidas e com possivelmente com maior acurácia nos resultados obtidos) [Cantu-Paz 1999].

Segundo [Breshears 2009], são quatro os passos da metodologia de programação com *threads*:

- análise: fase em que ocorre a identificação de possíveis regiões paralelas;
- projeto e implementação: fase de desenvolvimento do algoritmo paralelo;
- testes: fase para identificação e correção de erros;
- ajustes de desempenho: fase onde se identifica e remove de “gargalos“ de desempenho.

Este trabalho visa relatar os passos seguidos no processo de paralelização do algoritmo de evolução diferencial. Como aplicação de teste foi escolhido o problema do Despacho Econômico de Energia.

A seção 2 apresenta a descrição do problema a ser paralelizado e inclui o problema do despacho econômico de energia e o algoritmo de evolução diferencial utilizado para solucioná-lo. A seção 3 detalha os passos da paralelização implementada. Na seção 4 são apresentados os resultados e discussão dos testes realizados. Finalmente, na seção 5 são colocadas as conclusões e considerações finais, seguidas das referências bibliográficas.

2. Descrição do problema

O objetivo do problema do despacho econômico de energia, cujas características são complexas e altamente não lineares, é alocar a cada uma das unidades geradoras a quantidade de energia a ser produzida de forma que se tenham os custos de operação reduzidos, respeitando restrições de igualdade e desigualdade [Becerra and Coello Coello 2005]. A Evolução Diferencial é um algoritmo estocástico de otimização numérica [Feoktistov 2006].

Como base para este trabalho apresenta-se a implementação do problema do despacho econômico de energia utilizando a técnica evolutiva de evolução diferencial. O objetivo final é a implementação de uma solução paralela utilizando a notação OpenMP [Chapman et al. 2007] para a linguagem de programação C.

Existem alguns trabalhos recentes que implementam versões paralelas utilizando evolução diferencial: [Tasoulis et al. 2004], [Kwedlo and Bandurski 2006a] e [Kwedlo and Bandurski 2006b], reportando que bons resultados podem ser obtidos.

Nas seções seguintes o problema do despacho econômico é melhor detalhado, bem como a técnica de evolução diferencial.

2.1. Problema do Despacho Econômico

O Despacho Econômico é um dos mais importantes problemas a ser resolvido durante o planejamento e a operacionalização de um sistema de geração de energia elétrica [Park et al. 2006]. O objetivo do despacho econômico é minimizar o custo total de combustível de geradores de energia elétrica sujeito a restrições operacionais. A função objetivo desse problema pode ser formulada como:

$$\begin{gathered} \text{Minimizar F} = \sum_{j=1}^{n} F_j(P_j) \\ \text{sujeito a} \\ PD = \sum_{j=1}^{n} P_j \text{ e} \\ P_j^{min} \leq P_j \leq P_j^{max}, j = 1, \ldots, n \end{gathered} \tag{1}$$

onde $F_j(P_j)$ é a função que representa o custo do *jth* gerador (em \$/h), P_j é a potência de saída do *j-ésimo* gerador, *n* é o número de geradores no sistema, PD é a demanda total do sistema, P_j^{min} e P_j^{max} são, respectivamente, as saídas mínimas e máximas do *j-ésimo* gerador. A restrição de igualdade da Equação 1 é chamada de restrição de balanceamento de potência enquanto que as restrições de desigualdade são chamadas de restrições operacionais.

A função de custo de uma unidade geradora, sem considerar os efeitos de ponto de válvula, é dada por uma aproximação quadrática [Balamurugan and Subramanian 2007]:

$$F_j(P_j) = a_j * P_j^2 + b_j * P_j + c_j \tag{2}$$

onde P_j é a saída do *j-ésimo* gerador e a_j, b_j e c_j são coeficientes que determinam o custo de combustível do *j-ésimo* gerador.

Mas geradores com turbinas com múltiplas válvulas exibem uma variação muito maior das funções de custo devido ao efeito de ponto de válvula [Park et al. 2006]. O

efeito de ponto de válvula resulta em ondulações na curva de custo de combustível de um gerador provocada pela abertura de válvulas. A função de custo considerando o efeito das válvulas resulta num grau de não-linearidade maior. Então a Equação 2 deve ser substituída pela Equação 3, que considera os efeitos das válvulas. Matematicamente tem-se uma combinação de uma função quadrática e uma função senoidal:

$$F_j(P_j) = a_j * P_j^2 + b_j * P_j + c_j + |e_j * \sin(f_j * (P_j^{min} - P_j))| \quad (3)$$

onde P_j é a saída da *j-ésima* unidade geradora, a_j, b_j, c_j, e_j and f_j são os coeficientes da função de custo do *j-ésimo* gerador quando o efeito de ponto de válvula é considerado. Para este trabalho esta foi a função utilizada.

2.2. Evolução Diferencial

A Evolução Diferencial [Storn and Price 1997] é um algoritmo evolutivo para otimização de parâmetros de ponto flutuante que foi desenvolvido por Storn e Price em meados da década de noventa e se inicia criando uma população inicial escolhida aleatoriamente.

A idéia principal da evolução diferencial é gerar novos indivíduos, chamados vetores doadores, pela adição da diferença ponderada entre dois indivíduos aleatórios da população a um terceiro indivíduo. Esta é a operação de mutação. As componentes do indivíduo doador são misturadas com as componentes de um indivíduo escolhido aleatoriamente (chamado vetor alvo), para resultar em um vetor experimental. Este é o processo denotado cruzamento. Se o vetor experimental resultar um valor da função objetivo menor que o vetor alvo, então o vetor experimental substitui o vetor alvo na geração seguinte. Esta operação é chamada de seleção. O procedimento é finalizado através de algum critério de parada. A evolução diferencial trabalha com alguns parâmetros que são D (número de dimensões), NP (tamanho da população), CR (constante de cruzamento), F (peso do vetor de diferenças - constante de mutação) e MAX-GER (número máximo de gerações). O algoritmo é sumarizado a seguir.

```
1.  Inicializar parâmetros da Evolução Diferencial
2.  Inicializar aleatoriamente a população inicial
3.  Avaliar cada indivíduo da população inicial
4.  Repetir até que o critério de parada seja satisfeito
5.    Para cada indivíduo
6.       Selecionar aleatoriamente 3 indivíduos da população
7.       Aplicar operadores de diferenciação, mutação, crossover
8.       Comparar indivíduo com o sua versão experimental e
            selecionar o de menor custo para a nova população
9.       Avaliar o custo do indivíduo selecionado
10.   Fim Para cada indivíduo
11. Fim Repetir
```

O alvo da paralelização da técnica de evolução diferencial são as linhas 5 a 10 onde os operadores são aplicados a cada indivíduo.

O algoritmo de evolução diferencial aplicada ao problema do despacho econômico é mostrado a seguir.

```
  Inicializar parâmetros da Evolução Diferencial
  Inicializar aleatoriamente a população inicial dentro de
```

```
   limites inferiores e superiores para cada gerador
Reparar indivíduos que não atendem a demanda pretendida
Avaliar cada indivíduo de acordo com a formula de fitness
Repetir até que o critério de parada seja satisfeito
  Para cada indivíduo
     Selecionar aleatoriamente 3 indivíduos da população
     Aplicar operadores de diferenciação, mutação, crossover
     Reparar indivíduos que não atendem a demanda pretendida
    Comparar indivíduo com sua versão experimental e
        selecionar o de menor custo para a nova população
    Avaliar o custo do indivíduo selecionado
 Fim Para cada indivíduo
Fim Repetir
```

3. Descrição da solução do problema

Até chegar a versão paralela final, foram testadas algumas implementações. Utilizando a ferramenta de medida de performance *Intel VTune* foram obtidas as porcentagens (Tabela 1) de utilizacão do tempo do programa sequencial.

Nome da Função	Porcentagem de uso
aplicaOperadores	37,99%
avaliaIndividuo	15,28%
reparaIndividuo	14,69%
sin	13,29%
random	6,21%
Todas as outras funções	12,54%

Tabela 1. Informações da ferramenta *VTune* para a execução do programa sequencial.

O procedimento `AplicaOperadores` toma a maior parte do tempo da execução do programa, sendo justamente o foco da paralelização. Os procedimentos `AvaliaIndividuo` e `ReparaIndividuo` são chamadas dentro do procedimento `AplicaOperadores`. A função de seno (`sin`) é utilizada no cálculo da função de avaliação de aptidão (*fitness*). A função `random` é utilizada na geração da população inicial e no procedimento que aplica operadores aos indivíduos da população. Segue uma breve descrição da evolução conseguida.

A primeira versão de paralelização do problema, na versão de pseudo-código, é mostrada a seguir.

```
Inicializar parâmetros da Evolução Diferencial
Inicializar aleatoriamente a população inicial dentro de
   limites inferiores e superiores para cada gerador
Reparar indivíduos que não atendem a demanda pretendida
Avaliar cada indivíduo de acordo com a formula de fitness
Repetir até que o critério de parada seja satisfeito
   #pragma omp parallel
   {
    #pragma omp parallel for
    {
     Para cada indivíduo
```

```
          Selecionar aleatoriamente 3 indivíduos da população
          Aplicar os operadores de diferenciação,
             mutação, crossover
          Reparar indivíduos que não atendem a
             demanda pretendida
       Fim Para cada indivíduo
       #pragma omp single
        Avaliar população
       #pragma omp parallel for
        Comparar cada indivíduo com sua versão experimental e
           selecionar o de menor custo para a nova população
      }
     }
  Fim Repetir
```

A versão sequencial compilada com o `icc` executa em aproximadamente 2,4 segundos. Esta primeira versão paralela executa em aproximadamente 10 segundos.

A primeira alteração feita foi a troca da função que gera números aleatórios do C (`rand`) por uma função paralela (*LeapFrog*), o que reduziu o tempo de execução paralelo para aproximadamente 4 segundos.

A execução do programa paralelo com os `pragmas` definidos da seguinte forma:

```
#pragma omp parallel
 #pragma omp for
```

ao invés do uso da palavra `parallel` com o `pragma for`, reduziu o tempo de execução para aproximadamente 1,2 segundos, visto evitou a criação redundante de *threads*.

Algumas modificações que melhoraram o tempo de execução menos significativamente:

- Uso do compilador `icc` ao invés do `gcc`.
- Uso dos procedimentos como *inline*.
- Uso da biblioteca xSSSE3 na compilação.
- Gerar todos os números aleatórios necessários de uma vez e depois utilizá-los (*pool* de *randoms*). Esta técnica funcionou melhor do que gerar um número aleatório por vez.

Algumas tentativas que não melhoraram ou pioraram o tempo de execução:

- Paralelização da geração da população inicial. Como o tempo piorou, a geração da população inicial permaneceu sequencial.
- Fazer cópia local dos valores constantes utilizados no cálculo da função de aptidão. Como o tempo piorou, estes valores foram mantidos globais.
- Uso da função `memcpy` em alguns trechos de código.
- Trocar os tipos de *double* para *float*. Permaneceu o uso do *double*.
- Modificações usando a cláusula *schedule* do OpenMP.

Fez-se necessária também a substituição da função `abs` do C por uma função definida pelo usuário (denominada no código por `Absoluto`), devido a um problema de imprecisão de casas decimais o que resultou em erro no resultado final da produção de energia dos geradores. Esta modificação resultou em uma melhora, embora pouco significativa, no tempo de execução.

A ferramenta da Intel *Thread Checker*, que analisa código paralelo identificando pontos paralelizados problemáticos, foi aplicada ao código implementado e forneceu informações interessantes que ajudaram a identificar pontos de modificação. A Figura 1 mostra um trecho da saída do *Thread Checker* para o problema implementado (inicialmente foram reportados 151 erros). Neste caso são indicadas algumas corridas críticas.

```
_____________________________________________________________________________
|34|Write -> |Err|32|omp pa|Memory read of ind[] at "main.c":224  |"main|"main|
|  |Read     |or |61|rallel|conflicts with a prior memory write of|.c":1|.c":2|
|  |data-race|   |93|region|mutante[][] at "main.c":164 (flow     |64   |24   |
|  |         |   |  |      |dependence)                            |     |     |
_____________________________________________________________________________
|35|Read ->  |Err|32|omp pa|Memory write of mutante[][] at        |"main|"main|
|  |Write    |or |61|rallel|"main.c":164 conflicts with a prior   |.c":2|.c":1|
|  |data-race|   |93|region|memory read of ind[] at "main.c":224  |24   |64   |
|  |         |   |  |      |(anti dependence)                      |     |     |
_____________________________________________________________________________
|36|Write -> |Err|32|omp pa|Memory read of ind[] at "main.c":225  |"main|"main|
|  |Read     |or |61|rallel|conflicts with a prior memory write of|.c":1|.c":2|
|  |data-race|   |93|region|mutante[][] at "main.c":164 (flow     |64   |25   |
|  |         |   |  |      |dependence)                            |     |     |
_____________________________________________________________________________
|37|Read ->  |Err|32|omp pa|Memory write of mutante[][] at        |"main|"main|
|  |Write    |or |61|rallel|"main.c":164 conflicts with a prior   |.c":2|.c":1|
|  |data-race|   |93|region|memory read of ind[] at "main.c":225  |25   |64   |
|  |         |   |  |      |(anti dependence)                      |     |     |
_____________________________________________________________________________
```

Figura 1. Saída do *Thread Checker*.

A listagem 1 mostra o código em C paralelizado. Algumas modificações de código implementado também surtiram efeito e serão melhor explanadas a seguir.

Código 1. Código em C paralelizado.

```
1  inline void aplicaOperadores(double pop[][TAM_CROMOSSOMO], double
       fitness[], VSLStreamStatePtr* streams){
2      int i;
3      double mutante[NP][TAM_CROMOSSOMO], f_mutante[NP];;
4  #pragma omp parallel num_threads(NUM_THREADS) {
5      #pragma omp for
6      for (i = 0; i < NP; i++) { //Para cada individuo
7          int me = omp_get_thread_num();
8          int k, aleatorio_r0, aleatorio_r1, aleatorio_r2;
9          double r0[TAM_CROMOSSOMO], r1[TAM_CROMOSSOMO], r2[
               TAM_CROMOSSOMO];
10         int pool_randoms[8];
11         int pos_pool = 0;
12         viRngUniform(0, streams[me], 8, pool_randoms, 0, NP);
13         do {
14             //viRngUniform(0, streams[me], 1, &aleatorio_r0, 0, NP);
15             aleatorio_r0 = pool_randoms[pos_pool++];
16         } while (aleatorio_r0 == i);
17         for (k = 0; k < TAM_CROMOSSOMO; k++)
18             r0[k] = pop[aleatorio_r0][k];
19         //memcpy(r0, pop[aleatorio_r0], TAM_CROMOSSOMO*sizeof(double));
20         do {
21             //viRngUniform(0, streams[me], 1, &aleatorio_r1, 0, NP);
22             aleatorio_r1 = pool_randoms[pos_pool++];
23         } while (aleatorio_r1 == i || aleatorio_r1 == aleatorio_r0);
```

```
24          for (k = 0; k < TAM_CROMOSSOMO; k++)
25              r1[k] = pop[aleatorio_r1][k];
26          //memcpy(r1, pop[aleatorio_r1], TAM_CROMOSSOMO*sizeof(double));
27          do {
28              //viRngUniform(0, streams[me], 1, &aleatorio_r2, 0, NP);
29              aleatorio_r2 = pool_randoms[pos_pool++];
30          } while (aleatorio_r2 == i || aleatorio_r2 == aleatorio_r1 ||
                aleatorio_r2 == aleatorio_r0);
31          for (k = 0; k < TAM_CROMOSSOMO; k++)
32              r2[k] = pop[aleatorio_r2][k];
33          //memcpy(r0, pop[aleatorio_r0], TAM_CROMOSSOMO*sizeof(double));
34          //Aplicar diferenciacao, mutacao, crossover
35          double temp;
36          for (k = 0; k < TAM_CROMOSSOMO; k++){
37              vdRngUniform(0, streams[me], 1, &temp, 0.0, 1.0);
38              if (temp < CR)
39                  mutante[i][k] = r0[k] + F * (r1[k] - r2[k]);
40              else
41                  mutante[i][k] = pop[i][k];
42          }
43          reparaIndividuo(mutante[i], streams, me);
44          //#pragma omp critical
45          //avaliaIndividuo(i, f_mutante);
46      } //Fim Para cada individuo
47      //#pragma omp single
48      //avaliaPop(NP, f_mutante, mutante);
49      #pragma omp for
50      for(i = 0; i < NP; i++) {
51          int j;
52          double fitness_temp = 0.0;
53          for (j = 0; j < TAM_CROMOSSOMO; j++)
54              fitness_temp = fitness_temp + (a[j] * mutante[i][j] *
                    mutante[i][j]) + (b[j] * mutante[i][j]) + c[j] +
                    absoluto(e[j] * sin(f[j] * (limiteInferior[j] - mutante
                    [i][j])));
55          f_mutante[i] = fitness_temp;
56      }
57      #pragma omp for
58      for (i = 0; i < NP; i++) { //Para cada individuo
59      int k;
60       //Comparar individuo com sua versao experimental e selecionar o de
             menor custo para a nova populacao;
61          if (f_mutante[i] < fitness[i]) {
62              for (k = 0; k < TAM_CROMOSSOMO; k++)
63                  pop[i][k] = mutante[i][k];
64              //memcpy(pop[i], mutante[i], TAM_CROMOSSOMO*sizeof(double));
65              fitness[i] = f_mutante[i];
66          }
67      }
68  } //fim da secao paralela
69  }
```

Algumas observações sobre o código, baseadas nas informações fornecidas anteriormente nesta seção:

- A linha 12 mostra a criação de todos os números aleatórios que serão utilizados

(*pool* de randons) para cada *thread*. Sua utilização é feita nas linhas 15, 22 e 29. Anteriormente, cada valor aleatório era criado no momento em que era utilizado(conforme linhas comentadas 14, 21 e 28).

- Como as instruções das linhas 18, 25 e 32 são executadas repetidas vezes no código, foi cogitada a substituição do `for` correspondente a esta instrução pelo comando `memcpy`, com o intuito de melhor utilizar recursos e possibilitar uma melhora no tempo de execução. Tal medida não rendeu o resultado esperado.
- As linhas comentadas 44 e 45 mostram uma primeira versão da avaliação dos indivíduos mutados. Cada individuo era mutado, reparado e avaliado, sendo que a avaliação (por gravar no vetor de *fitness*) caracterizava uma região crítica. Para excluir esta região crítica, foi criado um procedimento de avaliação da população inteira de mutados (`AvaliaPop`), retirando a avaliação deste `for` paralelo (linha 48). Esta medida refletiu em ganho de tempo de execução. Posteriormente, o código foi trazido para a região paralela (linhas 50 a 56) o que melhorou ainda mais o desempenho.

Após estas modificações, a ferramenta *Thread Checker* verificou a implementação e não retornou nenhum ponto com problemas.

4. Resultados e Discussão

Esta seção sumariza os resultados dos testes realizados. Inicialmente, foram utilizados 1, 2, 4 e 8 processadores de uma máquina UMA *iseven* com 8 processaodres do *cluster* do laboratório de Sistemas Distribuídos da UTFPR (Universidade Tecnológica Federal do Paraná). Cada processador tem o modelo Intel(R) Core(TM) i7 CPU 860 2.80 GHz. A Tabela 2 apresenta os resultados obtidos para 100 execuções de cada configuração do problema: para 13, 40 e 80 geradores. A tabela sumariza a média do tempo de execução (em segundos) e o desvio padrão, respectivamente. Pode-se observar que ocorreu uma leve queda do tempo de execução conforme o número de *threads* aumentou.

	Número de geradores		
threads	**13**	**40**	**80**
1	0.890966 0.000630	2.331092 0.000661	4.493708 0.000370
2	0.471852 0.000365	1.202477 0.000319	2.288453 0.001309
4	0.454163 0.000304	1.068658 0.000291	2.030148 0.000143
8	0.241745 0.003405	0.616461 0.003299	1.083503 0.003350

Tabela 2. Média do tempo de execução e desvio padrão para máquina UMA.

Na Tabela 3 são apresentados os resultados de *speedup* para os testes.

Na Tabela 4 são apresentados os resultados de eficiência [Chapman et al. 2007], obtido dividindo-se o *speedup* pelo número de processadores.

Pode-se observar que a eficiência diminui conforme o número de *threads* aumenta, embora haja um pequeno aumento nos valores de *speedup*. Isto se explica pelo fato de

	Número de geradores		
threads	**13**	**40**	**80**
2	1,888231903	1,938575125	1,963644436
4	1,961775838	2,181326486	2,213487884
8	3,685561232	3,781410341	4,147388609

Tabela 3. Valores de *speedup* para máquina UMA.

	Número de geradores		
threads	**13**	**40**	**80**
2	94%	96%	98%
4	49%	54%	55%
8	46%	47%	51%

Tabela 4. Valores de eficiência para máquina UMA.

que com o aumento no número de *threads*, o tempo que o programa gasta executando a parte paralela diminui, sendo que o tempo gasto executando a parte sequencial, entretanto, permanece o mesmo. No entanto, visto que houve um certo ganho, os resultados podem ser considerados satisfatórios.

Para uma comparação informal, foram realizados testes em uma máquina NUMA da UFPR (Universidade Federal do Paraná). Cada processador tem o modelo Intel(R) Xeon(R) CPU E5345 2.33 GHz. A Tabela 5 sumariza os resultados do tempo de execução (em segundos), desvio padrão, *speedup* e eficiência para 80 geradores.

threads	Tempo e Desvio padrão	*Speedup*	Eficiência
1	5.858134 0.001034		
2	3.441427 0.001948	1,702239798	85%
4	2.009943 0.000614	2,91457718	72%
8	1.158785 0.001337	5,055410624	63%

Tabela 5. Resultados para 80 geradores usando máquina NUMA.

Percebe-se uma diferença significativa entre os resultados do programa paralelo em máquina UMA e NUMA. Isto se deve principalmente ao fato de que o código do programa esta otimizado para acesso uniforme a memória.

5. Conclusões

Este relatório técnico apresentou os resultados obtidos para a paralelização da técnica de evolução diferencial aplicada ao problema do despacho econômico.

O código paralelo mostrou bons resultados em relação ao tempo de execução do código sequencial. As principais alterações que refletiram na melhora do desempenho neste trabalho foram:

- Substituição da função `rand` sequencial do C para geração de números aleatórios por uma função paralela.
- Organização dos `pragmas` de forma a minimizar a criação e destruição de *threads*.
- Modificação de código de forma a eliminar regiões críticas.

Alguns pontos que contribuíram, ainda que de forma menos significativa foram: escolha do compilador (gcc/icc), uso da função *inline*e uso de bibliotecas de otimização na compilação e *pool* de *randoms*.

Vale ressaltar ainda que foram observadas algumas considerações durante a codificação, seguindo recomendações conforme [Chapman et al. 2007], dentre as quais pode-se citar:

- Percorrer os *loops* ao longo das linhas (e não das colunas) visando assegurar bom desempenho do sistema de memória.
- Diminuir ao máximo número de instruções dentro dos *loops*.
- Evitar o uso de barreiras.
- Maximizar regiões paralelas.
- Evitar a paralelização de *loops* internos.
- Analisar o balanceamento de carga entre as *threads*.
- Analisar as variáveis quanto a compartilhadas e privadas.

Dentre as dificuldades encontradas destaca-se a de conceber a solução do problema já pensando em paraleliza-lá. É importante a análise e o estudo de vários problemas e identificação de pontos paralelizáveis em diferentes situações de forma a treinar o raciocínio e programação computacionais paralelos.

Particularmente, este trabalho possibilitou ganho de experiência na programação utilizando OpenMP, bem como na codificação paralela para técnica evolutiva que eh um dos temas da minha pesquisa de doutorado. Todo o processo de aprendizado que envolveu o desenvolvimento deste trabalho foi de muito proveito.

Como trabalhos futuros, pode-se projetar e implementar uma versão do problema utilizando CUDA, estabelecendo um comparativo com a versão para OpenMP. Além disso, pode-se otimizar o código para funcionar em uma máquina com arquitetura NUMA para se verificar o comportamento. Finalmente, pretende-se testar a troca do problema do Despacho Econômico de Energia pelo da Predição da Estrutura de Proteínas (problema foco da minha pesquisa) utilizando evolução diferencial a fim de confirmar se os bons resultados se mantém utilizando um problema mais complexo e com mais variáveis envolvidas.

Referências

Balamurugan, R. and Subramanian, S. (2007). Self-adaptive differential evolution based power economic dispatch of generators with valve-point effects and multiple fuel options. *International Journal of Computer Science and Engineering*, 1(1).

Becerra, R. L. and Coello Coello, C. A. (2005). Optimization with constraints using a cultured differential evolution approach. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 27–34, New York, NY, USA. ACM.

Breshears, C. (2009). *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly, Sebastopol, CA.

Cantu-Paz, E. (1999). *Designing efficient and accurate parallel genetic algorithms (parallel algorithms)*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA. Adviser Goldberg, David E.

Chapman, B., Jost, G., and Pas, R. v. d. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.

Feoktistov, V. (2006). *Differential Evolution: In Search of Solutions (Springer Optimization and Its Applications)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Humenay, E., Tarjan, D., and Skadron, K. (2007). Impact of process variations on multicore performance symmetry. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1653–1658, San Jose, CA, USA. EDA Consortium.

Kwedlo, W. and Bandurski, K. (2006a). A parallel differential evolution algorithm a parallel differential evolution algorithm. *Parallel Computing in Electrical Engineering, International Conference on*, 0:319–324.

Kwedlo, W. and Bandurski, K. (2006b). A parallel differential evolution algorithm a parallel differential evolution algorithm. In *PARELEC '06: Proceedings of the international symposium on Parallel Computing in Electrical Engineering*, pages 319–324, Washington, DC, USA. IEEE Computer Society.

Park, J., Jeong, Y., and Lee, W. (2006). An improved particle swarm optimization for economic dispatch problems with non-smooth cost functions. In *IEEE Power Engineering Society General Meeting*, Montreal, Que.

Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group.

Storn, R. and Price, K. (1997). Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, 11(4):341–359.

Tasoulis, D. K., Pavlidis, N. G., Plagianakos, V. P., and Vrahatis, M. N. (2004). Parallel differential evolution. In *Congress on Evolutionary Computation (CEC 2004)*, Portland, Oregon.