

Laboratório Nacional de Computação Científica  
Relatório Técnico LNCC

# **Modelagem Multi-dimensional e inversão de dados eletromagnéticos (EM) em uma estação de trabalho multicore utilizando OpenMP ou processamento paralelo heterogêneo**

Rafael Lourenço Stanislau

Co-autores: Tiziano Labruzzo (Schlumberger), Roberto Souto (LNCC), Carla Osthoff (LNCC), e Andrea Zerilli (Schlumberger)

Petrópolis, RJ - Brasil  
Agosto de 2017

# Listas de figuras

Figura 1 – Arquitetura dos processadores multi-core utilizados . . . . .	7
Figura 2 – Parâmetros do modelo definidos no arquivo ModelCan2D_invC2_ini.mcm	8
Figura 3 – Trecho de código em Newton.f90, no qual é escolhida a forma de execução: modelo direto (IFLAG=0), modelo inverso (IFLAG=1,2). O valor de IFLAG é definido pelo valor do parâmetro de entrada Niter. . . . .	9
Figura 4 – Domínio do estudo de caso, contendo 11 pontos de coleta de dados geofísicos, distribuídos em intervalos igualmente espaçados de 1000 metros, partindo da distância de 10000 metros até 20000 metros . . . . .	9
Figura 5 – Árvore de chamada das rotinas na execução do problema direto . . . . .	11
Figura 6 – Estrutura da matriz esparsa $A$ , de dimensão $n = 80083$ , contendo 716713 elementos não-nulos complexos ( $nnz = 716713$ ) . . . . .	12
Figura 7 – Trecho do código fonte de lancos3e.f90, que originalmente desativa a execução multi-thread. . . . .	12
Figura 8 – Trecho do código fonte de lancos3e.f90, onde é liberada a execução multi-thread. . . . .	12
Figura 9 – Trecho de código em lancos3e.f90, que realiza chamada do solver da PARDISO, que realiza a fase de fill-in reduction e fatoração simbólica (ipphase=11). . . . .	14
Figura 10 – Trecho de código em lancos3e.f90, que realiza chamada do solver da PARDISO, que realiza a fase de fatoração numérica (ipphase=22). . . . .	14
Figura 11 – Trecho de código em lancos3e.f90, que realiza chamada do solver da PARDISO, que realiza a fase de cálculo da solução do sistema (ipphase=33). .	14
Figura 12 – Árvore de chamada das rotinas na execução do problema inverso . . . . .	17
Figura 13 – Código-fonte alterado, onde é retirada a condição de corrida observada na variável <b>IJ</b> , que contém o índice do array <b>AtA</b> . . . . .	21
Figura 14 – Nova subrotina criada em substituição à DLLT0, para a realização da fatoração de Cholesky . . . . .	23
Figura 15 – Subrotina original do código para fatoração de Cholesky . . . . .	24
Figura 16 – Trecho de código onde é paralelizado um dos três hotspots da subrotina COMPUTE_RES_IHRS . . . . .	25
Figura 17 – . . . . .	26
Figura 18 – Trecho de código onde é paralelizado a chamada da subrotina DERTAU() .	26
Figura 19 – Erro residual no sítio 10000 . . . . .	29
Figura 20 – Erro residual no sítio 11000 . . . . .	30
Figura 21 – Erro residual no sítio 12000 . . . . .	30
Figura 22 – Erro residual no sítio 13000 . . . . .	31

Figura 23 – Erro residual no sítio 14000 . . . . .	31
Figura 24 – Erro residual no sítio 15000 . . . . .	32
Figura 25 – Erro residual no sítio 16000 . . . . .	32
Figura 26 – Erro residual no sítio 17000 . . . . .	33
Figura 27 – Erro residual no sítio 18000 . . . . .	33
Figura 28 – Erro residual no sítio 19000 . . . . .	34
Figura 29 – Erro residual no sítio 20000 . . . . .	34

# Listas de tabelas

Tabela 1 – Perfil de desempenho do modelo direto mostrando as 5 funções que mais consomem processamento . . . . .	10
Tabela 2 – Tempos do modelo direto para diferentes números de threads . . . . .	13
Tabela 3 – Desempenho computacional do modelo direto . . . . .	13
Tabela 4 – Desempenho computacional da região paralela do modelo direto . . . .	13
Tabela 5 – Desempenho computacional da fatoração numérica da matriz A dos coeficientes, realizada pelo solver da biblioteca PARDISO (phase=22) .	15
Tabela 6 – Desempenho computacional para o cálculo da solução do sistema linear, após feita a fatoração da matriz A dos coeficientes, realizado pelo solver da biblioteca PARDISO (phase=33) . . . . .	15
Tabela 7 – Desempenho computacional conjunto da fatoração numérica e do cálculo da solução, no solver da biblioteca PARDISO (phase=22+phase=33) .	15
Tabela 8 – Perfil de desempenho do modelo inverso (NIter=2), obtido com a ferramenta Intel Vtune . . . . .	16
Tabela 9 – Tempos (elapsed time) do modelo inverso para diferentes números de threads . . . . .	18
Tabela 10 – Desempenho computacional do modelo inverso . . . . .	18
Tabela 11 – Desempenho computacional da região paralela do modelo inverso . . .	18
Tabela 12 – Erro residual na segunda iteração do modelo inverso (NIter=2), quando executado com diferentes números de threads (arquivo ModelCan2D_invC2_ini.l2c) *ver figuras no Anexo A, para o caso com 16 threads . . . . .	19
Tabela 13 – Perfil de desempenho do modelo inverso com multifreqüência mostrando as 5 funções que mais consomem processamento . . . . .	20
Tabela 14 – Comparação de desempenho entre as opções da cláusula SCHEDULE no trecho de multiplicação de matrizes . . . . .	21
Tabela 15 – Desempenho computacional do cálculo de multiplicação de matrizes com OpenMP com fator D0 . . . . .	21
Tabela 16 – Desempenho computacional total do cálculo jacobiano com OpenMP .	22

# Sumário

<b>1</b>	<b>Introdução</b>	<b>5</b>
<b>2</b>	<b>Modelo direto</b>	<b>7</b>
2.1	Desempenho do solver da biblioteca PARDISO no modelo direto	13
<b>3</b>	<b>Modelo inverso</b>	<b>16</b>
<b>4</b>	<b>Modelo inverso com multifrequências</b>	<b>20</b>
<b>5</b>	<b>Considerações finais</b>	<b>27</b>
5.1	Trabalhos futuros	27
<b>Anexos</b>		<b>28</b>
<b>ANEXO A Erros residuais para os resultados de inversão obtidos com o uso de 16 threads no solver da biblioteca PARDISO</b>		<b>29</b>

# 1 Introdução

Nos últimos dez anos, métodos de baixa frequência eletromagnética tem recebido muito interesse para exploração em áreas complexas pela sua capacidade de detectar elementos que possam se associar com reservas de hidrocarbonetos e para complementar medidas sísmicas.

Dois tipos de métodos eletromagnéticos são aplicados sendo o primeiro uma medida ativa chamada de método eletromagnético de fonte controlada (CSEM)(Johansen et al., 2005) e o segundo um método magnetotelúrico (MT) guiado pelas fontes naturais (Vozoff, 1991). Ambos métodos medem vetores de campo magnético e elétrico. Com o objetivo de extrair o máximo de informação dos dados do CSEM e MT aplica-se uma abordagem de inversão não linear na sua interpretação.

Dois tipos de abordagens para inversão estão disponíveis, sendo a primeira abordagem conhecida como algoritmo de inversão baseado em imagem (IBI), que tem demonstrado capacidade de retornar imagens de condutividade razoavelmente boas, entretanto, os contornos reconstruídos e os valores de condutividade das anomalias da imagem não são usualmente resolvidos quantitativamente. A segunda abordagem é o algoritmo de inversão baseado em estrutura (SBI, structure-based inversion), que usa primordialmente a informação geométrica para reduzir a quantidade de parâmetros desconhecidos a melhorar a qualidade da imagem de condutividade reconstruída.

A inversão multidimensional eletromagnética implementada no SBI é uma tarefa difícil e computacionalmente intensa, fazendo dela uma área ativamente pesquisada. A otimização requer linearização do operador direto que representa as equações de Maxwell, atingido pela introdução da matrix derivada de Fréchet e obtendo a atualização do modelo fatorando a matriz de Hessian aproximada regularizada, uma matriz densa, real e simétrica, usando o método LDLT (Golub and Van Loan, 1996), (ambas as tarefas intensas e demoradas).

Por outro lado, o desenvolvimento tecnológico e científico na área de computação de alto desempenho tem gerado o aparecimento de diversas novas arquiteturas de processadores que possibilitam a execução paralela de algoritmos computacionalmente intensos, tais como as arquiteturas multicore, manycore e many integrated core (MIC) entre outros.

O plano de trabalho apresentado teve como objetivo reestruturar e paralelizar os algoritmos da abordagem SBI para uma ambiente de computação de alto desempenho para processadores multicore, através do modelo de programação OpenMP. As principais áreas de desenvolvimento do trabalho estão relacionadas com o laço de frequência contendo a modelagem direta, chamado de modelo direto, cálculos residuais e das derivadas de Fréchet e a fatoração da matriz de Hessian, chamados de modelo inverso.

Conforme os resultados apresentados neste relatório, mostramos que através da paralelização, da reestruturação de código e da implementação de bibliotecas otimizadas,

foi possível diminuir de forma significativa o tempo de execução da aplicação quando executada de forma paralela em diversos núcleos de um processador multicore em relação ao código original, que executa em um ambiente com apenas um núcleo de processamento.

Na seção 2 apresentamos os resultados relacionados ao modelo direto. Na seção 3 mostramos os resultado resultado do desempenho computacional do modelo inverso com mono frequência e na seção 4 com o modelo multifrequência e na seção 5 as nossas considerações finais.

## 2 Modelo direto

O modelo direto calcula a resposta geofísica para um dado modelo geológico, enquanto que o modelo inverso estima a estrutura geológica a partir dos dados geofísicos observados.

Esta etapa do trabalho consiste em identificar as rotinas que mais consomem tempo de processamento, através da obtenção do perfil de desempenho serial e paralelo para as rodadas dos modelos direto e inverso. Foi utilizada para este fim a ferramenta de avaliação de desempenho VTune, disponível no pacote Intel Parallel Studio.

A Schlumberger disponibilizou uma estação de trabalho contendo 2 CPU Intel Xeon E5-2687W (8-core, 3.10GHz), com um total de 181 Gbytes de RAM. A arquitetura detalhada destas CPUs multi-core é mostrada na Figura 1. A máquina possui também uma GPU NVIDIA Tesla C2075, para fins de processamento de alguns trechos da aplicação.



Figura 1 – Arquitetura dos processadores multi-core utilizados

A Figura 2 apresenta o conteúdo do arquivo de entrada denominado `ModelCan2D_invC2_ini.mcm`, com os parâmetros do modelo direto, assim como a estrutura do estudo de caso a ser

estudado.

O arquivo de entrada fornece os parâmetros de execução. O parâmetro NIter define o número de iterações a serem executadas. Para NIter=0, será executado apenas o modelo direto (inverse model), para execução também do modelo inverso (inverse model), deve ser utilizar um valor acima de um, que corresponde ao número de iterações.

A Figura 3, apresenta o código gerado após a leitura do parâmetros fornecidos pelo arquivo de entrada e apresenta a forma como será executada a aplicação, se como modelo direto (IFLAG=0), ou como modelo inverso (IFLAG=1,2).

```
"Model updated by IX_SC2 v2.35 Date:02/07/14 Time:18:51"
&SCS2D
Header(1)=' Line',
Header(3)='Data from',
StnBeg=0.00,StnEnd=30000.00,StnPerDipole=200.00,
DpLength=200.0,LengthUnits='m',DpAzimuth=90.0,
DataMT=F,DataCSEM=T,DataSeismicRefrac=F,DataSeismicReflec=F,ActiveDomain='CSEM',
InversionMT=F,InversionCSEM=T,InversionSeismicRefrac=F,InversionSeismicReflec=F,
SurveyTypeCSEM='Scalar',TxTypeCSEM='EDipole',Environment='Marine',
XNodes=151,ZNodes=22,PolygonCentralPadXNodes=301,PolygonCentralPadZNodes=64,
WaterElevation=0.00000,
WaterLayerResistivity(1)=0.305000,
AnisotropyEM=T,JointMultiDataInversionEM=T,
NIter=0,UseAmp=T,UsePhase=F,FarFieldFloor=1.0000E-2,
ErrFloorPercent=0.00000,MinOffsetEM=1000.00,MaxOffsetEM=7500.00,
BathymetricCorrection=100.000,GlobalErrorFloor=T,
ElectricHorDataNoiseFloor=0.200E-15,ElectricVertDataNoiseFloor=0.100E-14,
MagneticHorDataNoiseFloor=0.700E-12,MagneticVertDataNoiseFloor=0.300E-11,
SetInversionModel=3,NLayers=7,InvertOnEM=1,InversionType=0,
AssignModelAngle=22.00,ActiveRho=V,
AssignCMPGathers=-2,ResampledSourceDistance=100.000,
HResSmth=0.100,HResdpW=0.010,HResdxW=1.000,HResdzW=1.000,
VResSmth=0.100,VResdpW=0.010,VResdxW=1.000,VResdzW=1.000,
SaveAtEachIteration=T,
```

Figura 2 – Parâmetros do modelo definidos no arquivo ModelCan2D\_invC2\_ini.mcm

O domínio especificado no arquivo de parâmetros, é mostrado na Figura 4, que contém 11 pontos de coleta de dados geofísicos, distribuídos em intervalos igualmente espaçados de 1000 metros, partindo da distância de 10000 metros até 20000 metros. Esta figura foi gerada pelo código de interface gráfica, após ler os dados fornecidos pelo arquivo de entrada da Figura 2.

---

```

select case (IFLAG)
case (0) ! Just computing Forward
    return
case (1) ! update residuals and vector gradient
    call compute_dFV(m_bAnisotropy)
case (2) ! update residuals, vector gradient and Hessian
    call compute_dFV(m_bAnisotropy)
    call compute_QNA_Hessian(RDMAX)
end select

```

---

Figura 3 – Trecho de código em Newton.f90, no qual é escolhida a forma de execução: modelo direto (IFLAG=0), modelo inverso (IFLAG=1,2). O valor de IFLAG é definido pelo valor do parâmetro de entrada Niter.

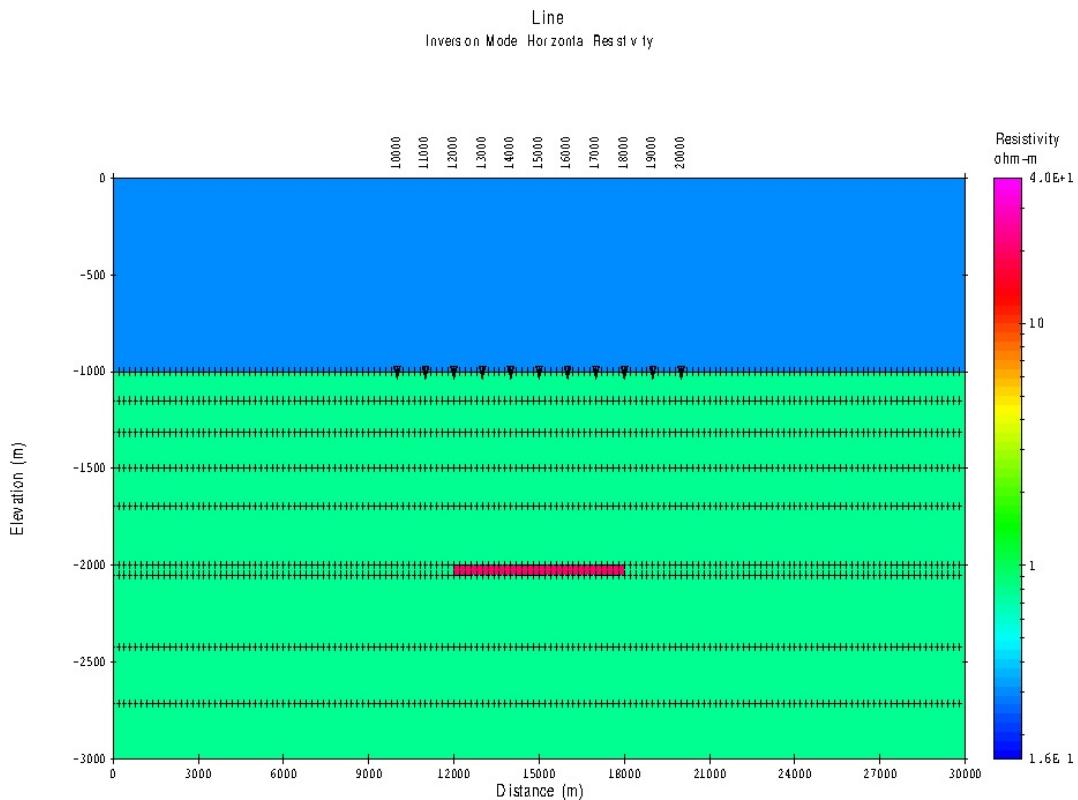


Figura 4 – Domínio do estudo de caso, contendo 11 pontos de coleta de dados geofísicos, distribuídos em intervalos igualmente espaçados de 1000 metros, partindo da distância de 10000 metros até 20000 metros

A Figura 5 mostra a árvore de chamada das principais funções acionadas durante a execução do modelo direto, e a Tabela 1 mostra as 5 rotinas que mais consomem tempo de processamento. O tempo total de processamento do modelo direto foi de 100.16 segundos, sendo que destacam-se duas funções da biblioteca MKL PARDISO: blkslv\_ll\_cmplx e blkl\_ll\_cmplx.

Durante a execução do modelo direto, estas funções são utilizadas na resolução de

sistemas de equações lineares, denotados por  $Ax = b$ , onde  $A$  é a matriz dos coeficientes dos sistemas. Esta matriz é esparsa e simétrica, contendo elementos não-nulos do tipo complexo.

Para o estudo de caso adotado, o sistema possui 80083 incógnitas ( $n = 80083$ ), com a matriz  $A$  contendo 716713 elementos complexos não nulos ( $nnz = 716713$ ). A estrutura esparsa da matriz dos coeficientes é mostrada na Figura 6.

A função `blk_ll_cmplx` é responsável por efetuar a fatoração numérica da matriz  $A$ , enquanto que a função `blkslv_ll_cmplx`, após feita a fatoração, realiza a etapa de substituição e retro-substituição, para calcular os valores das incógnitas de  $x$  no sistema  $Ax = b$ . Juntas, estas duas funções gastam cerca de 86% do tempo total de processamento.

A biblioteca MKL PARDISO provê suporte para execução multi-thread para as funções `blkslv_ll_cmplx` e `blk_ll_cmplx`. No entanto, a execução paralela estava desativada originalmente no código-fonte do modelo direto (arquivo `lancos3e.f90`), conforme é mostrado na Figura 7. Portanto, a estratégia de otimização no processamento, consistiu basicamente em habilitar a execução paralela com threads para o solver da MKL PARDISO, como mostrado na Figura 8.

Tabela 1 – Perfil de desempenho do modelo direto mostrando as 5 funções que mais consomem processamento

Função	Tempo(s)	Tempo relativo(%)
MKL_PARDISO_blkslv_ll_cmplx	73.53	73.42%
MKL_PARDISO_blk_ll_cmplx	12.89	12.87%
<code>sig_</code>	2.38	2.37%
MKL_PARDISO_c_psol_fwgath_pardiso	1.54	1.54%
<code>_intel_new_memset</code>	1.53	1.53%
Others	8.29	8.28%
Total	100.16	100%

A Tabela 2 contém o tempo de processamento do modelo direto, para a execução sequencial com uma thread, e para as execuções paralelas com 2, 4, 8 e 16 threads. O tempo total de processamento é também separado em tempo da região efetivamente paralelizada, do tempo da região que roda sequencialmente. Importante ressaltar que a região paralelizada é referente apenas às funções utilizadas da biblioteca PARDISO, para a resolução do sistema linear de equações do modelo direto.

O tempo da região sequencial oscila em torno de 10.5 segundos, para qualquer número de threads utilizado. Logicamente, conforme é reduzido tempo da região paralelizada com o aumento do número de threads, proporcionalmente o tempo da região sequencial aumenta de importância com relação ao tempo total de processamento.

A Tabela 3 e a Tabela 4 apresentam, respectivamente, o desempenho computacional obtido pelo modelo direto como um todo, e o obtido somente pela região paralelizada. É alcançada uma boa eficiência paralela com o modelo direto – em torno de 74% - para o uso de até 4 threads. A partir do emprego de 8 threads, a escalabilidade reduz-se de forma considerável.

Ao analisarmos o desempenho obtido somente na região paralelizada, percebe-se que a escalabilidade é muito boa com o uso de até 8 threads, atingindo neste caso uma eficiência paralela de 88%. Entretanto, com o uso de 16 threads, a eficiência decai para 56%.

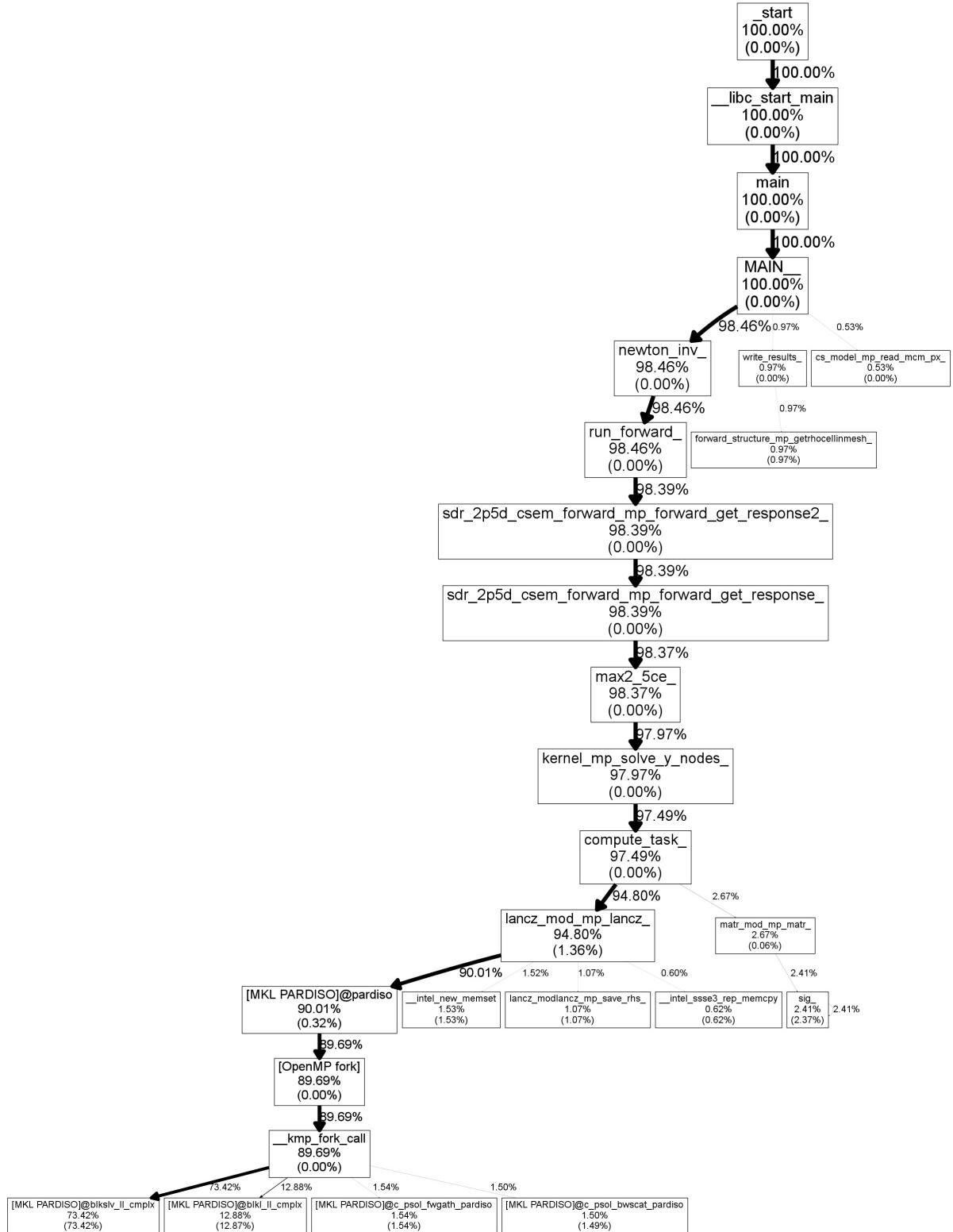


Figura 5 – Árvore de chamada das rotinas na execução do problema direto

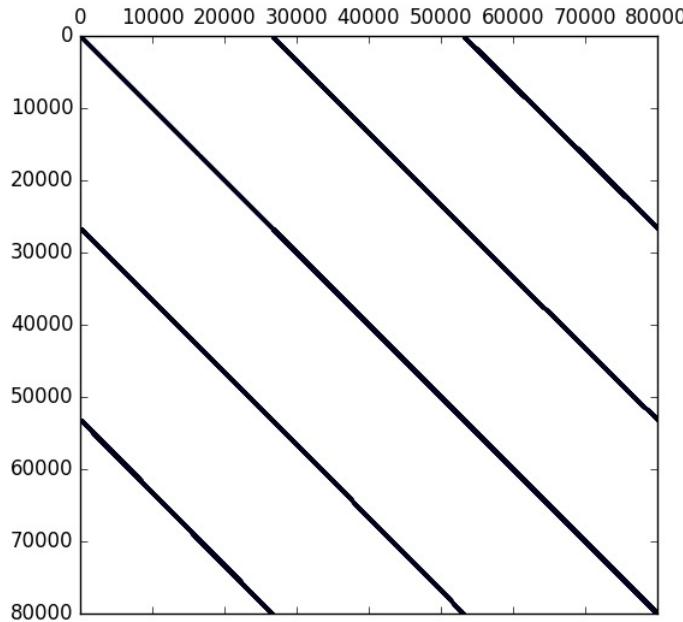


Figura 6 – Estrutura da matriz esparsa  $A$ , de dimensão  $n = 80083$ , contendo 716713 elementos não-nulos complexos ( $nnz = 716713$ )

---

```

! disable multithreading to avoid numerical differences
call mkl_set_num_threads( 1 )

!      factorize matrix
iphase=22
call pardiso(ipt,maxfct,mnum,mtype,iphase,n,a,ia,ja,iperm,
            nrhspardiso,iparm,msglvl,ddum,ddum,ierror)
call check_error(ierror)

```

---

Figura 7 – Trecho do código fonte de lancos3e.f90, que originalmente desativa a execução multi-thread.

---

```

! enable multithreading
! call mkl_set_num_threads( 1 )

!      factorize matrix
iphase=22
call pardiso(ipt,maxfct,mnum,mtype,iphase,n,a,ia,ja,iperm,
            nrhspardiso,iparm,msglvl,ddum,ddum,ierror)
call check_error(ierror)

```

---

Figura 8 – Trecho do código fonte de lancos3e.f90, onde é liberada a execução multi-thread.

Tabela 2 – Tempos do modelo direto para diferentes números de threads

Threads	Tempo(s)	Tempo região paralela(s)	Tempo relativo região paralela	Tempo serial(s)
1	100.28	89.59	89%	10.69
2	56.54	46.17	82%	10.37
4	34	23.32	69%	10.68
8	23.84	12.78	54%	11.07
16	20.57	9.91	48%	10.66

Tabela 3 – Desempenho computacional do modelo direto

Threads	Tempo(s)	Speedup	Eficiência(%)
1	100.28	1	100%
2	56.54	1.77	89%
4	34	2.95	74%
8	23.84	4.21	53%
16	20.57	4.87	30%

Tabela 4 – Desempenho computacional da região paralela do modelo direto

Threads	Tempo(s)	Speedup	Eficiência(%)
1	89.59	1.00	100%
2	46.17	1.94	97%
4	23.32	3.84	96%
8	12.78	7.01	88%
16	9.91	9.04	56%

## 2.1 Desempenho do solver da biblioteca PARDISO no modelo direto

O solver da biblioteca PARDISO é executado no modelo direto de 3 formas distintas, dependendo da etapa, ou fase, de resolução do sistema. Na primeira fase (phase=11), é realizada a reordenação de linhas e colunas das matriz dos coeficientes, a fim de minimizar a inclusão de elementos não-nulos (fill-in inclusion) durante a fatoração da matriz A, dos coeficientes do sistema linear de equações. Esta reordenação é feita por método desenvolvido na biblioteca METIS (<http://glaros.dtc.umn.edu/gkhome/views/metis>), que emprega particionamento de grafos.

Nesta fase é realizada também a fatoração simbólica, onde define-se então qual a posição dos elementos não-nulos das matrizes fatoradas. Na Figura 9 é mostrado o trecho onde é feita a chamada do solver da PARDISO, que realiza a fase de fill-in reduction e fatoração simbólica.

---

```

iparm(1) = 0
!iparm(2) = 0 ! use minimum degree fill-in reducing
!iparm(2) = 2 ! use metis nested dissection fill-in reducing
iparm(2) = 3 ! use parallel metis nested dissection fill-in reducing

iphase=11
tInicial = omp_get_wtime()
call pardiso(ippt,maxfct,mnum,mtype,iphase,n,a,ia,ja,iperm,
            nrhspardiso,iparm,msglvl,ddum,ddum,ierror)
tFinal = omp_get_wtime()
tTotal11 = tFinal - tInicial
print*, 'n, ne, Analysis Time (phase=11): ', n, ne, tTotal11
call check_error(ierror)

```

---

Figura 9 – Trecho de código em lancos3e.f90, que realiza chamada do solver da PARDISO, que realiza a fase de fill-in reduction e fatoração simbólica (iphase=11).

O valor numérico do destes elementos não-nulos, é definido na fase seguinte (phase=22), onde ocorre a fatoração numérica (Figura 10). Após realizada a fatoração numérica, ocorre então na etapa posterior (phase=33) o cálculo da solução do sistema (Figura 11).

---

```

iphase=22
tInicial = omp_get_wtime()
call pardiso(ippt,maxfct,mnum,mtype,iphase,n,a,ia,ja,iperm,
            nrhspardiso,iparm,msglvl,ddum,ddum,ierror)
tFinal = omp_get_wtime()
tTotal22 = tFinal - tInicial
print*, 'Factoring Time (phase=22): ', tTotal22
call check_error(ierror)

```

---

Figura 10 – Trecho de código em lancos3e.f90, que realiza chamada do solver da PARDISO, que realiza a fase de fatoração numérica (iphase=22).

---

```

iphase=33
tInicial = omp_get_wtime()
call pardiso(ippt,maxfct,mnum,mtype,iphase,n,a,ia,ja,iperm,
            nrhspardiso,iparm,msglvl,ddum,ddum,ierror)
tFinal = omp_get_wtime()
tTotal22 = tFinal - tInicial
print*, 'Substitution Time (phase=33): ', tTotal22
call check_error(ierror)

```

---

Figura 11 – Trecho de código em lancos3e.f90, que realiza chamada do solver da PARDISO, que realiza a fase de cálculo da solução do sistema (iphase=33).

Durante a execução do modelo direto, o solver da biblioteca PARDISO é chamado uma única vez na fase de reordenação e de fatoração simbólica, e chamado 37 vezes para fatoração numérica e para o cálculo da solução do sistema. Os tempos apresentados nas tabelas a seguir, são os tempos médios para as 37 execuções.

Nas Tabelas 5, 6 e 7 são mostrados, respectivamente, o desempenho computacional do solver da PARDISO nas fases de fatoração da matriz (phase=22), de cálculo da solução (phase=33) e de forma conjunta ambas as fases ( phase=22 + phase=33 ).

Embora o custo computacional da fase de cálculo da solução seja maior do que a fase de fatoração, esta etapa possui uma boa escalabilidade até mesmo com o uso de 16 threads, alcançando uma eficiência paralela de 71,6%. Já a fase de fatoração numérica, apresenta boa eficiência somente com o uso de até 4 threads (70%).

De modo geral, somando os tempos das duas fases, têm-se uma boa escalabilidade até 8 threads (86.7%), e uma razoável eficiência paralela com 16 threads (64.5%)

Tabela 5 – Desempenho computacional da fatoração numérica da matriz A dos coeficientes, realizada pelo solver da biblioteca PARDISO (phase=22)

<b>Threads</b>	<b>Tempo(s)</b>	<b>Speedup</b>	<b>Eficiência(%)</b>
1	0.354	1.00	100.0%
2	0.203	1.75	87.3%
4	0.126	2.80	70.0%
8	0.077	4.62	57.7%
16	0.054	6.56	41.0%

Tabela 6 – Desempenho computacional para o cálculo da solução do sistema linear, após feita a fatoração da matriz A dos coeficientes, realizado pelo solver da biblioteca PARDISO (phase=33)

<b>Threads</b>	<b>Tempo(s)</b>	<b>Speedup</b>	<b>Eficiência(%)</b>
1	2.066	1.00	100.0%
2	1.075	1.92	96.1%
4	0.517	4.00	99.9%
8	0.027	7.59	94.8%
16	0.180	11.45	71.6%

Tabela 7 – Desempenho computacional conjunto da fatoração numérica e do cálculo da solução, no solver da biblioteca PARDISO (phase=22+phase=33)

<b>Threads</b>	<b>Tempo(s)</b>	<b>Speedup</b>	<b>Eficiência(%)</b>
1	2.419	1.00	100.0%
2	1.277	1.89	94.7%
4	0.643	3.76	94.0%
8	0.349	6.93	86.7%
16	0.234	10.32	64.5%

### 3 Modelo inverso

Apresentamos neste capítulo, o resultado do desempenho computacional do modelo inverso, configurando-se agora o parâmetro de execução NIter=2.

A Figura 12 mostra a árvore de chamada das funções executadas no modelo inverso, e a Tabela 8 mostra os cinco primeiros hotspots da execução do modelo inverso. Assim como no modelo direto, o modelo inverso foi executado utilizando do suporte multi-thread disponibilizado pela biblioteca PARDISO. Como resultado, o Intel Vtune identificou que as funções presentes na biblioteca PARDISO, blkslv\_ll\_cmplx e blkl\_ll\_cmplx, novamente, tal como no modelo direto, são os principais hotspots, com cerca de 80% do tempo total de processamento.

Tabela 8 – Perfil de desempenho do modelo inverso (NIter=2), obtido com a ferramenta Intel Vtune

Função	Tempo(s)	Tempo relativo(%)
MKL_PARDISO_bklslv_ll_cmplx	768.49	69.41%
MKL_PARDISO_bkll_ll_cmplx	128.77	11.63%
mp_compute_res_source_	32.98	2.98%
__intel_new_memset	25.51	2.30%
sig_	23.63	2.13%
Others	127.76	11.54%
Total	1107.13	100%

A Tabela 9 mostra o tempo de processamento do modelo inverso, para a execução sequencial com uma thread, e para as execuções paralelas com 2, 4, 8 e 16 threads. Assim como realizado no modelo direto, o tempo total de processamento do modelo inverso é também separado em tempo da região que roda sequencialmente e região efetivamente paralelizada. Ressalta-se novamente que a região paralelizada é referente apenas às funções utilizadas da biblioteca PARDISO, para a resolução do sistema linear de equações do modelo direto.

Aqui observa-se uma maior oscilação no tempo de processamento da região sequencial, variando entre 154 e 188 segundos, para diferentes número de threads utilizadas. Conforme reduz-se tempo da região paralelizada com o aumento do número de threads, proporcionalmente o tempo da região sequencial aumenta de importância com relação ao tempo total de processamento.

A Tabela 10 apresenta o desempenho computacional obtido pelo modelo inverso como um todo. Ao analisarmos o desempenho obtido somente na região paralelizada, percebe-se que a escalabilidade é muito boa com o uso de até 2 threads, atingindo neste caso uma eficiência paralela de 78%. Entretanto, já a partir do uso de 4 threads, a eficiência decai para abaixo de 60%, chegando a somente 30% com 16 threads.

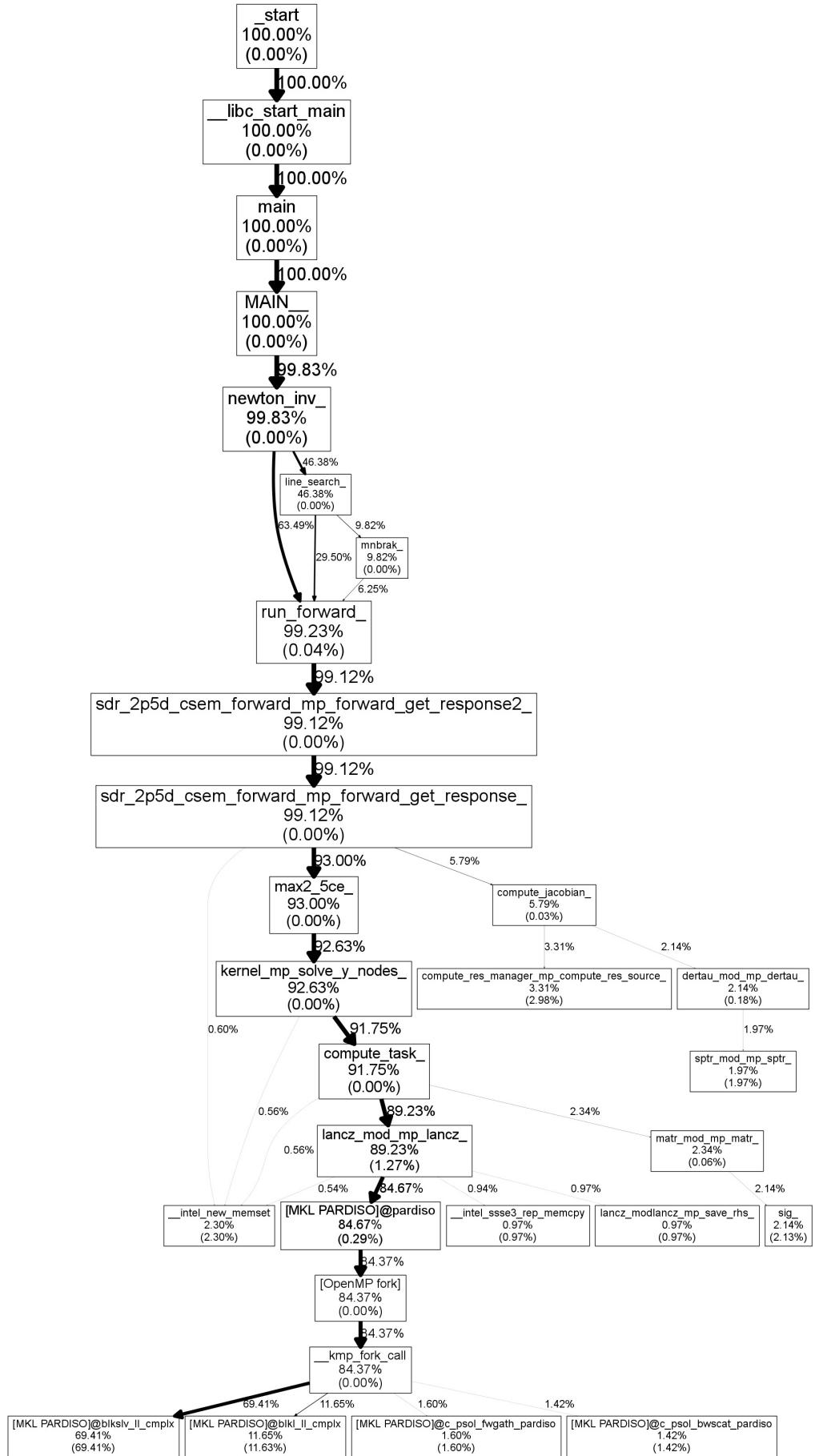


Figura 12 – Árvore de chamada das rotinas na execução do problema inverso

No entanto, esta perda de eficiência paralela poderia ser ainda mais acentuada, se não tivesse ocorrido um fato bastante incomum, observado através dos tempos de processamento somente da região paralelizada, mostrados na Tabela 11. Tem-se aqui um resultado de ganho de processamento paralelo superlinear, ou seja, o ganho é maior que o esperado. O speed-up alcançado com 8 threads, é maior do que este número de threads utilizadas.

De 4 para 8 threads, há uma significativa redução no tempo de processamento. A fim de melhor entender, e explicar corretamente este comportamento, é necessário que seja feita uma análise mais detalhada da execução paralela. Como hipótese, é provável que isto decorra do modo de atribuição das threads entre os núcleos da CPU multi-core (ver Figura 1).

Tabela 9 – Tempos (elapsed time) do modelo inverso para diferentes números de threads

Threads	Tempo(s)	Tempo região paralela(s)	Tempo relativo região paralela	Tempo serial(s)
1	1108.48	937.88	85%	170.60
2	706.99	528.55	75%	178.44
4	477.33	288.95	61%	188.38
8	248.05	93.99	38%	154.06
16	238.16	76.64	32%	161.52

Tabela 10 – Desempenho computacional do modelo inverso

Threads	Tempo(s)	Speedup	Eficiência(%)
1	1108.48	1.00	100%
2	706.99	1.57	78%
4	477.33	2.32	58%
8	248.05	4.47	56%
16	238.16	4.65	29%

Tabela 11 – Desempenho computacional da região paralela do modelo inverso

Threads	Tempo(s)	Speedup	Eficiência(%)
1	937.88	1.00	100%
2	528.55	1.77	89%
4	288.95	3.25	81%
8	93.99	9.98	125%
16	76.64	12.24	76%

A fim de validar a correção dos resultados gerados com a execução paralela, a Tabela 12 apresenta o erro residual da inversão realizada, para todos os números de threads utilizados. Em todos os casos, o erro residual encontra-se dentro um limiar aceitável, onde o erro ficou sempre abaixo de 5%.

Para corroborar ainda mais esta observação, no Anexo A foram acrescentadas figuras contendo gráfico comparando a curva de inversão obtida (em verde), com a curva da solução exata (em vermelho). Para todas os casos, as duas curvas estão bem próximas, indicando o acerto das inversões alcançadas.

Tabela 12 – Erro residual na segunda iteração do modelo inverso (NIter=2), quando executado com diferentes números de threads (arquivo ModelCan2D\_invC2\_ini.l2c)  
 \*ver figuras no Anexo A, para o caso com 16 threads

<b>Iteração 2</b>	<b>Threads</b>				
	<b>1</b>	<b>2</b>	<b>4</b>	<b>8</b>	<b>16*</b>
RMS MODEL CONSTRAINTS RESIDUAL	1.04%	1.04%	1.34%	1.02%	1.01%
RMS DATA CONSTRAINTS RESIDUAL	3.24%	3.24%	4.09%	2.79%	2.76%
RMS INVERSION RESIDUAL	1.82%	1.82%	2.31%	1.62%	1.61%

## 4 Modelo inverso com multifrequências

Uma vez que o estudo de caso avaliado anteriormente não identificou grande relevância no tempo de execução das rotinas referentes ao processo de inversão, criou-se um novo modelo, mais robusto, com multifrequências com a finalidade de observar seu comportamento.

Através do perfil de desempenho adquirido deste novo estudo de caso encontrou-se sub-rotinas do processo de inversão com alto tempo de execução, dentre elas destacam-se as rotinas: COMPUTE\_QNA\_HESSIAN, DLLT0 e COMPUTE\_RES\_IRHS, como pode ser visto na Tabela 13. Ao contrário do que vimos no modelo observado inicialmente, o processo de inversão consome mais de 90% do tempo de execução.

Tabela 13 – Perfil de desempenho do modelo inverso com multifreqüência mostrando as 5 funções que mais consomem processamento

Função	Tempo(s)
COMPUTE_QNA_HESSIAN	900.947
COMPUTE_RES_MANAGER_mp_COMPUTE_RES_IRHS	876.052
RltTryEnterCriticalSection	231.656
SPTR_MOD_mp_SPTR	177.523
DLLT0	176.200

Iniciou-se o processo de paralelização a partir da sub-rotina COMPUTE\_QNA\_HESSIAN (Newton.f90) pois ela realiza a operação de multiplicação de matrizes e chamada da função DLLT0. A abordagem foi na utilização de diretivas de compilação OpenMP, tal como é mostrado na Figura 13.

Numa implementação inicial, foi observado que havia uma condição de corrida (*race condition*) das *threads* que atualizam o valor da variável *IJ*, que armazena o índice do array AtA.

Foi necessário então refazer o cálculo de atualização de *IJ*, de maneira que o índice obtido fosse único e exclusivo para cada thread. Criou-se o índice  $IJ = ((J - 1) * J)/2 + I$  para substituir o original do código  $IJ = IJ + 1$ .

A cláusula SCHEDULE possui opções que permitem realizar a divisão de carga de trabalho de formas diferentes. Para este trabalho avaliou-se as opções: STATIC, DYNAMIC e GUIDED. Na STATIC a carga cada iteração do loop é distribuída para os threads de forma contígua. Na DYNAMIC as iterações são divididas entre os threads e caso haja iterações remanescentes, as mesmas são alocadas pelo primeiro thread a ficar disponível. A opção GUIDED possui uma distribuição parecida com a realizada na opção DYNAMIC, com a diferença que se caso houver iterações remanescentes, será feita uma nova distribuição dessas iterações entre os threads.

---

```

1 !$OMP PARALLEL PRIVATE(J, ASUM, IJ) NUM_THREADS(16)
2 !$OMP DO SCHEDULE(GUIDED)
3 DO J = 1, NP
4   DO I = 1, J
5     !IJ = IJ + 1 !occurs here a thread race condiction
6     !replaced by an exclusive value for each thread
7     IJ = ((J - 1) * J) / 2 + I
8     ASUM = 0.0
9     DO K = 1, ND
10       ASUM = ASUM + AJ(K, I) * Wd(K) * AJ(K, J)
11   END DO
12
13
14 !$OMP END DO
15 !$OMP END PARALLEL

```

---

Figura 13 – Código-fonte alterado, onde é retirada a condição de corrida observada na variável `IJ`, que contém o índice do array `AtA`

A Tabela 14 mostra a comparação de desempenho após testes realizando a utilização das três opções da cláusula SCHEDULE citadas anteriormente. A opção GUIDED conquistou um desempenho superior se comparado à opção STATIC e praticamente igual se comparado à opção DYNAMIC.

Tabela 14 – Comparação de desempenho entre as opções da cláusula SCHEDULE no trecho de multiplicação de matrizes

Opção	Tempo(s)
STATIC	559
DYNAMIC	301
GUIDED	299

Para uma melhor divisão dos elementos da matrizes entre as *threads* criadas, como mostrado anteriormente, foi utilizada a cláusula SCHEDULE(GUIDED), tal como consta no código-fonte da Figura 13.

Conforme resultados mostrados na Tabela 15, obteve-se um ganho de desempenho quase linear. Utilizando 16 *threads*, número referente a quantidade de núcleos computacionais da máquina, se conseguiu um ganho de 15.28 vezes com uma eficiência de 95%.

Tabela 15 – Desempenho computacional do cálculo de multiplicação de matrizes com OpenMP com fator D0

threads	Tempo(s)	Speedup	Eficiência(%)
1	4571	1.00	100%
2	2283	2.00	100%
4	1157	3.95	98%
8	591	7.73	96%
16	299	15.28	95%

Tabela 16 – Desempenho computacional total do cálculo jacobiano com OpenMP

<i>threads</i>	Tempo(s)	Speedup	Eficiência(%)
1	62	1.00	100%
2	30	2.06	103%
4	15	3.87	96%
8	8	7.75	96%
16	4	15.5	96%

Ainda código-fonte do arquivo Newton.f90, foi implementada a sub-rotina DLLT02 (Figura 14) em substituição à DLLT0 (Figura 15). Como basicamente a sub-rotina DLLT0 realizava a fatoração de Cholesky, percebeu-se que era possível a utilização de bibliotecas que resolvem esta operação.

Após pesquisas, encontrou-se a função SPORTF() presente na biblioteca MKL da Intel. Com a substituição e execução desta nova sub-rotina, notou-se um desempenho bem superior. O tempo de execução foi reduzido de 3097 para 8 segundos, tendo sido obtido um ganho de desempenho de 387 vezes.

Em relação a sub-rotina COMPUTE\_RES\_IRHS (compute\_res.f90), a qual faz parte do cálculo jacobiano, foram paralelizados três laços que realizam a mesma operação de multiplicação de matrizes.

A diferença consiste no fato de que apenas a variável *l0* assume um valor diferente para cada laço, podendo ser 1,2 ou 3. O código mostrado na Figura 16 apresenta um dos 3 trechos paralelizados, foram utilizadas as mesmas diretivas e cláusulas OpenMP.

A paralelização dos três laços da sub-rotina COMPUTE\_RES\_IHRS não apresentou um significativo ganho de desempenho, uma vez que o tempo de execução foi reduzido de 62 para 25 segundos, ao serem utilizadas 16 *threads*. No entanto, notou-se que era possível extrair mais desempenho da parte do código correspondente ao cálculo jacobiano.

Foi identificado um laço da sub-rotina COMPUTE\_JACOBIAN(compute\_jacobian.f90) onde era possível a paralelização. Conforme mostrado na Figura 18, este laço realiza a chamada da sub-rotina DERTAU e, após implementação das diretivas OpenMP, conseguiu-se obter uma melhora no desempenho, como pode ser visto na Tabela 16.

Com todas as paralelizações realizadas, o tempo total do cálculo jacobiano baixou de 62 segundos para 4 segundos utilizando 16 *threads*. Isso representa um ganho de desempenho de 15.50 vezes em relação ao código original, com uma eficiência de 96%.

---

```

1 SUBROUTINE DLLT02( N, NN2, A, IERR )
2 !-----
3 ! Performs the Cholesky decomposition
4 !-----
5 ! Arguments:
6 INTEGER, INTENT(IN) :: N           ! dimension of A
7 INTEGER, INTENT(IN) :: NN2         ! length of compact storage array
8 REAL, INTENT(INOUT) :: A(NN2)     ! Lower triangle of N by N matrix
9
10 ! Local variables:
11 INTEGER I, J, IJ, INFO
12 REAL,ALLOCATABLE :: AtA2(:,:)
13
14 ALLOCATE(AtA2(N,N))
15
16 IERR=0
17 IJ=0
18 AtA2 = 0.0
19 DO I = 1,N
20     DO J = 1,I
21         IJ = IJ + 1
22         AtA2(J,I) = A(IJ)
23         AtA2(I,J) = A(IJ)
24     END DO
25 END DO
26
27 CALL MKL_SET_NUM_THREADS(OMP_GET_MAX_THREADS())
28 CALL SPOTRF('U', N, AtA2, N, INFO)
29
30 IJ=0
31 I=0
32 DO I=1,N
33     AtA2(I,I) = 1.0 / (ATA2(I,I))
34 END DO
35
36 IJ=0
37 DO I = 1,N
38     DO J = 1,I
39         IJ = IJ + 1
40         A(IJ) = AtA2(J,I)
41     END DO
42 END DO
43 DEALLOCATE(AtA2)
44
45 END SUBROUTINE DLLT02

```

---

Figura 14 – Nova subrotina criada em substituição à DLLT0, para a realização da fatoração de Cholesky

---

```

1 SUBROUTINE DLLTO( N, NN2, A, IERR )
2 !-----
3 ! Performs the Cholesky decomposition  $A = L^t \cdot L$ . Adapted from Martin,
4 ! Peters, & Wilkinson, 1965, Symmetric decomposition of positive
5 ! definite martrix, Numer. Math., v7, p362-383.
6 !-----
7 ! Arguments:
8 INTEGER, INTENT(IN) :: N      ! dimension of A
9 INTEGER, INTENT(IN) :: NN2    ! length of compact storage array
10 REAL, INTENT(INOUT) :: A(NN2) ! Lower triangle of N by N matrix
11 ! A is the lower triangle of a symmetric positive definite matrix
12 ! stored in compact form, row by row. On output A holds the
13 ! lower triangular matrix L. Reciprocals of diagonal elements are
14 ! stored rather than the diagonal elements themselves.
15 INTEGER, INTENT(OUT) :: IERR ! error flag, 0=no error
16
17 ! Local variables:
18 INTEGER I, J, K, P, Q, R
19 REAL     DSUM
20 !-----
21 IERR = 0
22 P = 0
23 DO I = 1,N
24   Q = P + 1
25   R = 0
26   DO J = 1,I
27     DSUM = A(P+1)
28     DO K = Q,P
29       R = R + 1
30       DSUM = DSUM - A(K)*A(R)
31   END DO
32   R = R + 1
33   P = P + 1
34   IF( I==J ) THEN
35     IF( DSUM<=0.0 ) THEN ! Matrix is not positive definite.
36       write(*,*), 'ERROR: Matrix not positive definite in Cholesky decomp S/R DLLT.'
37       IERR = 1
38       A(P) = 0.0
39       GOTO 990
40   ELSE
41     A(P) = 1.0/SQRT(DSUM)
42   END IF
43   ELSE
44     A(P) = DSUM*A(R)
45   END IF
46 END DO
47 END DO
48
49 990 CONTINUE
50 RETURN
51 END SUBROUTINE DLLTO

```

---

Figura 15 – Subrotina original do código para fatoração de Cholesky

---

```

1  l0=1
2  !$OMP PARALLEL PRIVATE(l0, NODE, k0, J,I,K,RESY_INDEX) NUM_THREADS(16)
3  !$OMP DO SCHEDULE(GUIDED)
4  do i0=xbegin,xend+1
5      i = i0 - xbegin + 1
6      ! laço over my slices and compute a partial field into res
7      do node = 1, num_solved_local
8          do k0=zbegin,zend+1
9              k = k0 - zbegin + 1
10             resy_index = (i0-1)*nz2+k0-1
11             do j=2,ny1
12             !#ifdef DEBUG_BUILD
13             !       if (allnode(i0,j,k0,l0,nx,ny,nz)) then
14             !#endiff
15                 res(j,k,i,l0) = res(j,k,i,l0) + svy_all(j,node_results(node)%node_id)&
16                 & * node_results(node)%resy(it)%ptr(resy_index,irhs) * 1d-9
17             !#ifdef DEBUG_BUILD
18             !       end if
19             !#endiff
20                 end do
21             end do
22         end do
23     end do
24  !$OMP END DO
25  !$OMP END PARALLEL

```

---

Figura 16 – Trecho de código onde é paralelizado um dos três hotspots da subrotina COMPUTE\_RES\_IHRS

Figura 17 –

---

```

1 !$OMP PARALLEL PRIVATE(IX, IZ, DER, DERXY, DERZ) NUM_THREADS(16)
2 !$OMP DO SCHEDULE(GUIDED)
3 do ix=1,xukn
4   do iz=1,zukn
5     call dertau(opt%xbegin-1+ix, opt%zbegin-1+iz, &
6       compcomb, irhsq, der, derxy, derz)
7     if(psizes.gt.max_nrecu) then
8       jac_r(tel_local,(IX-1)*ZUKN+IZ) = &
9       dreal(opt%ck(j,i,f)*derxy*c_norm) * model_params%fon
10      jac_i(tel_local,(IX-1)*ZUKN+IZ) = &
11      dimag(opt%ck(j,i,f)*derxy*c_norm) * model_params%fon
12      jac_r(tel_local,xukn*zukn+(ix-1)*zukn+iz) = &
13      dreal(opt%ck(j,i,f)*derz*c_norm) * model_params%fon
14      jac_i(tel_local,xukn*zukn+(ix-1)*zukn+iz) = &
15      dimag(opt%ck(j,i,f)*derz*c_norm) * model_params%fon
16    else
17      jac(          tel,(ix-1)*zukn+iz) &
18      = dreal(opt%ck(j,i,f)*derxy*c_norm) * model_params%fon
19      jac(ncount_jac + tel,(ix-1)*zukn+iz) &
20      = dimag(opt%ck(j,i,f)*derxy*c_norm) * model_params%fon
21      jac(          tel,xukn*zukn+(ix-1)*zukn+iz) &
22      = dreal(opt%ck(j,i,f)*derz*c_norm) * model_params%fon
23      jac(ncount_jac + tel,xukn*zukn+(ix-1)*zukn+iz) &
24      = dimag(opt%ck(j,i,f)*derz*c_norm) * model_params%fon
25    endif
26  enddo
27 enddo
28 !$OMP END DO
29 !$OMP END PARALLEL

```

---

Figura 18 – Trecho de código onde é paralelizado a chamada da subrotina DERTAU()

# 5 Considerações finais

Os resultados de desempenho computacional apresentados neste relatório, evidenciam a relevância que a tarefa de resolução do sistema linear esparso tem para a aplicação. Por este motivo, é muito importante que o solver utilizado na resolução dos sistemas lineares, seja executado no menor tempo possível.

A aplicação utiliza o solver PARDISO, que faz parte da biblioteca Intel MKL, e implementa método direto de resolução de sistemas. Como estratégia trivial para ganho de desempenho, foi habilitada a execução paralela com threads para o solver PARDISO. Esta opção estava desabilitada anteriormente no código. Essa alteração permitiu um ganho, em cerca, de 1.4 vezes, ou 40%, do tempo total da execução do modelo direto.

Adquirido o estudo de caso com multifrequências foi possível identificar sub-rotinas que correspondiam ao processo de inversão. Após processo de paralelização com diretivas OpenMP, foi possível um ganho de 15.50 vezes, em 16 threads que representa o numero total de núcleos disponíveis na máquina, no que diz respeito às operações realizadas durante o cálculo jacobiano. Já no trecho que efetua a multiplicação de matrizes, conquistou-se um desempenho de 15.28 vezes, quando executado com 16 threads.

Com a implementação da sub-rotina DLLT02 que adota biblioteca otimizada para fatoração de Cholesky, obteve-se um ganho de 387 vezes em relação a sub-rotina original que realizava esta operação.

Ao final da execução completa de uma iteração com fator d0, obteve-se um ganho próximo de 15 vezes (14.85), com 16 threads, em relação à execução do código original.

## 5.1 Trabalhos futuros

No que diz respeito ao modelo direto, propõe-se o estudo e utilização de outro solver de equações lineares semelhante à PARDISO, sobretudo que utilize processamento heterogêneo com GPU, com a finalidade de obter melhora no desempenho atual.

Uma vez paralelizado, o trecho que efetua a multiplicação de matrizes ainda possui tempo de execução significativo. Em princípio, busca-se o estudo de uma biblioteca otimizada que possa substituir essa operação, assim como feito na fatoração de Cholesky.

Adicionalmente, realizar a execução da aplicação em uma máquina com número maior de núcleos, para realizar testes de escalabilidade do código atual.

## Anexos

# ANEXO A – Erros residuais para os resultados de inversão obtidos com o uso de 16 threads no solver da biblioteca PARDISO

Figura 19 – Erro residual no sítio 10000

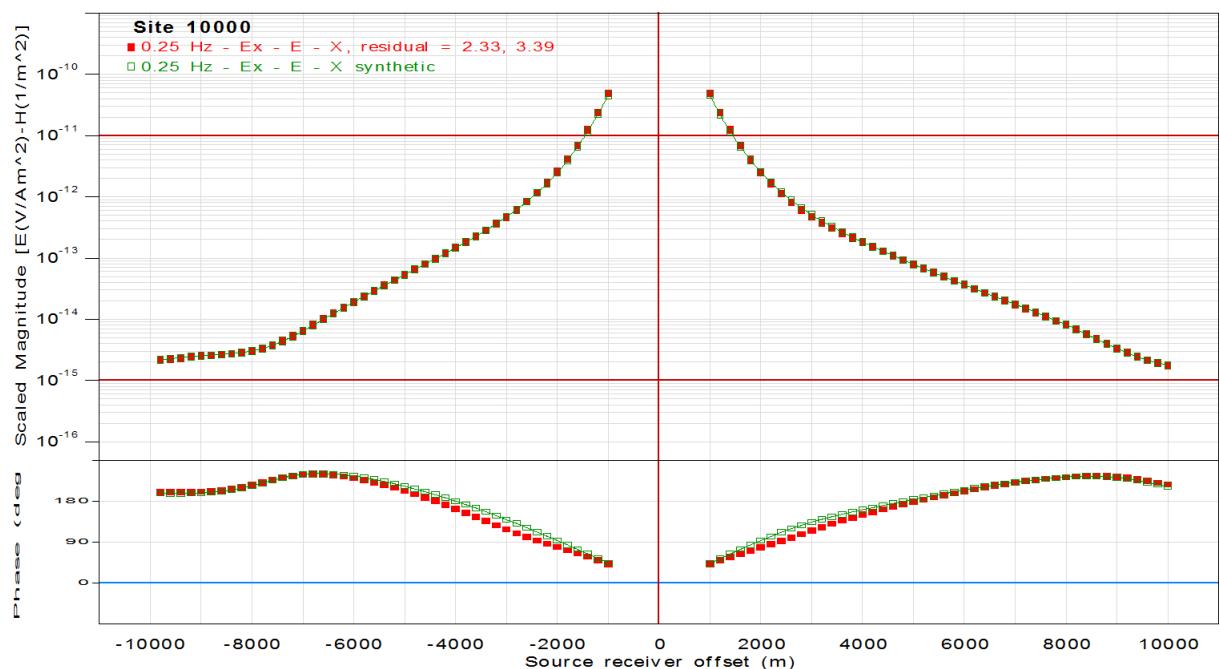


Figura 20 – Erro residual no sítio 11000

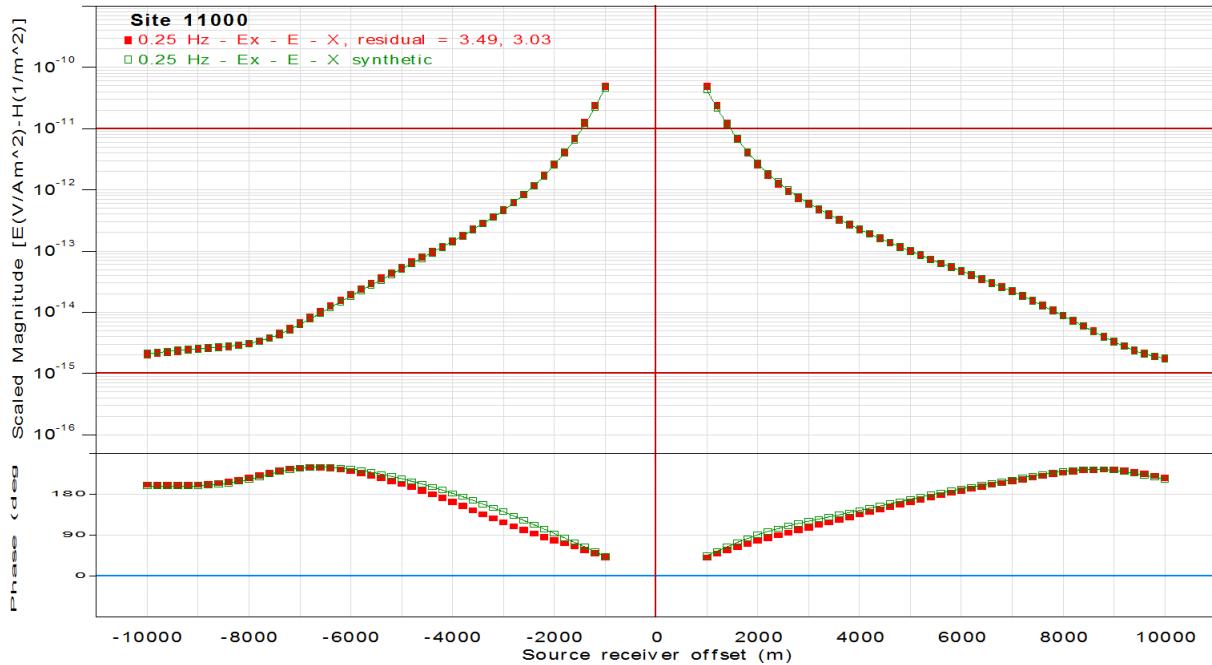


Figura 21 – Erro residual no sítio 12000

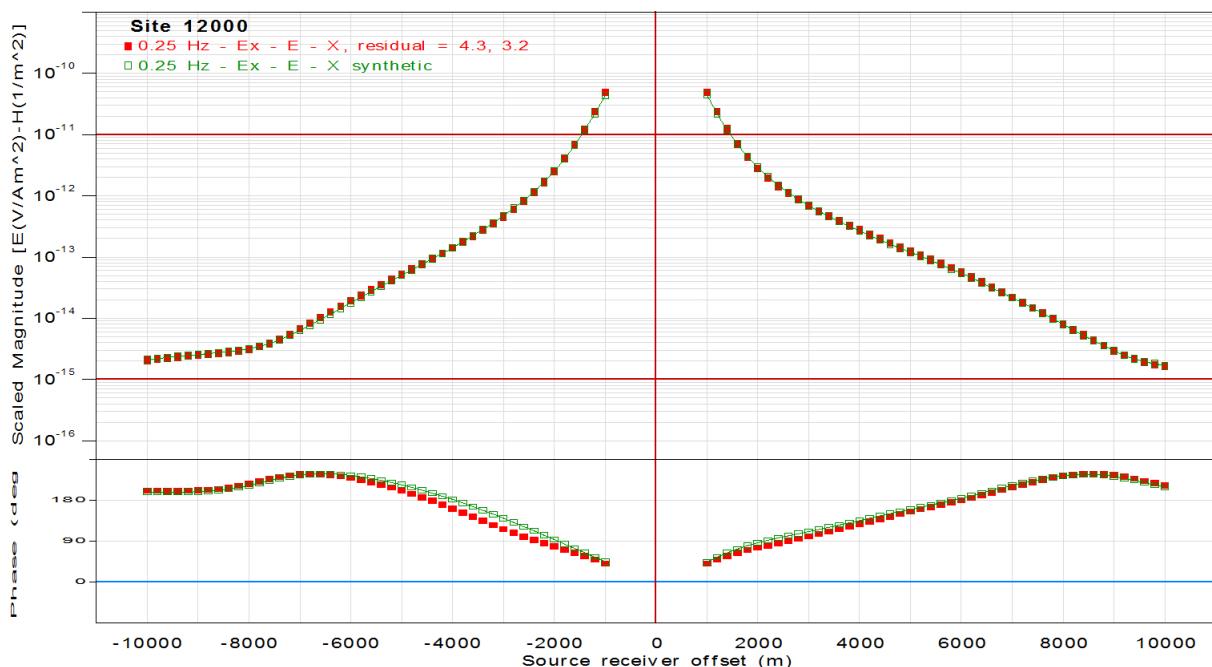


Figura 22 – Erro residual no sítio 13000

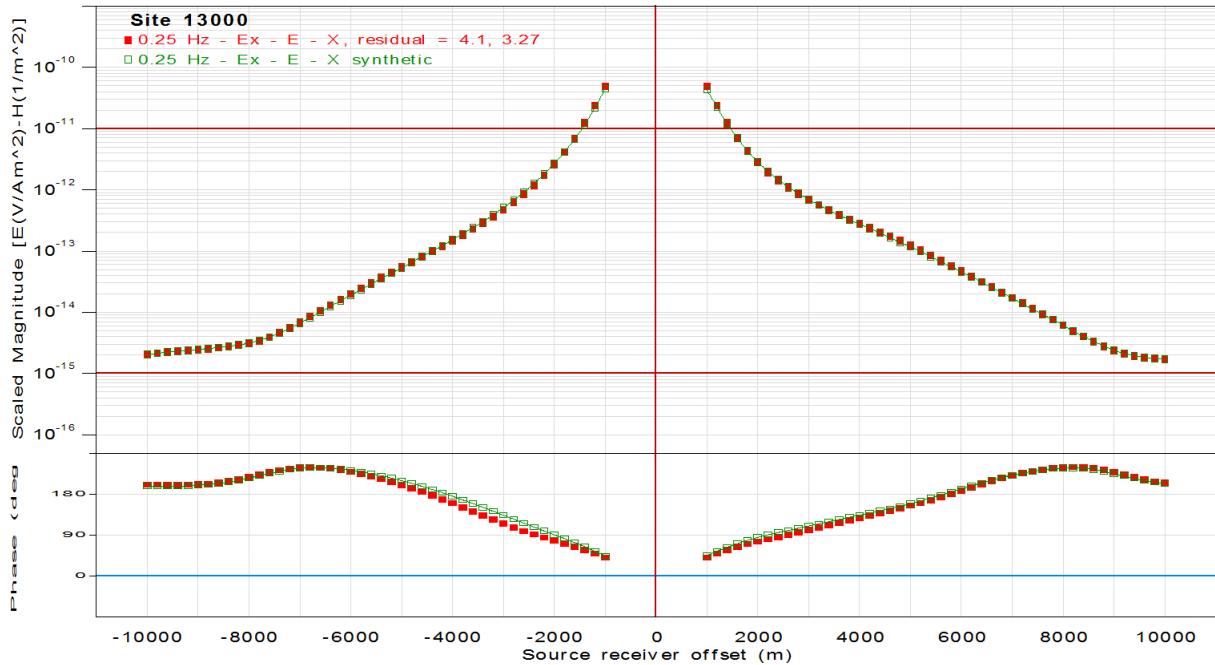


Figura 23 – Erro residual no sítio 14000

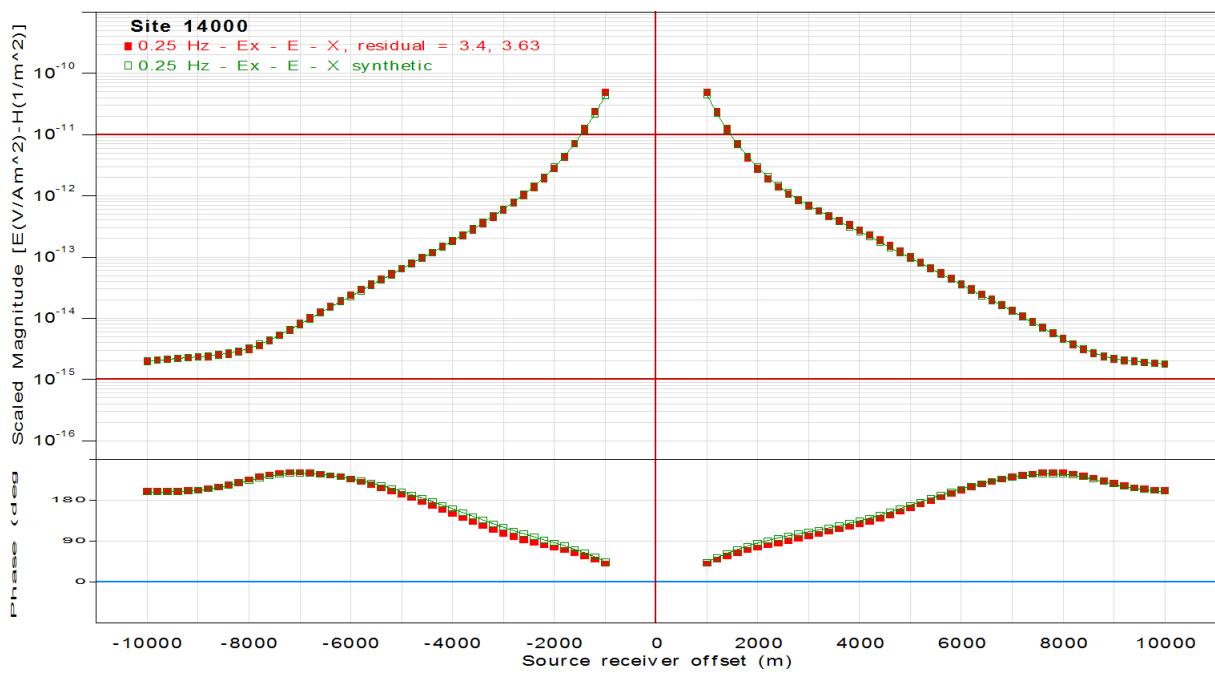


Figura 24 – Erro residual no sítio 15000

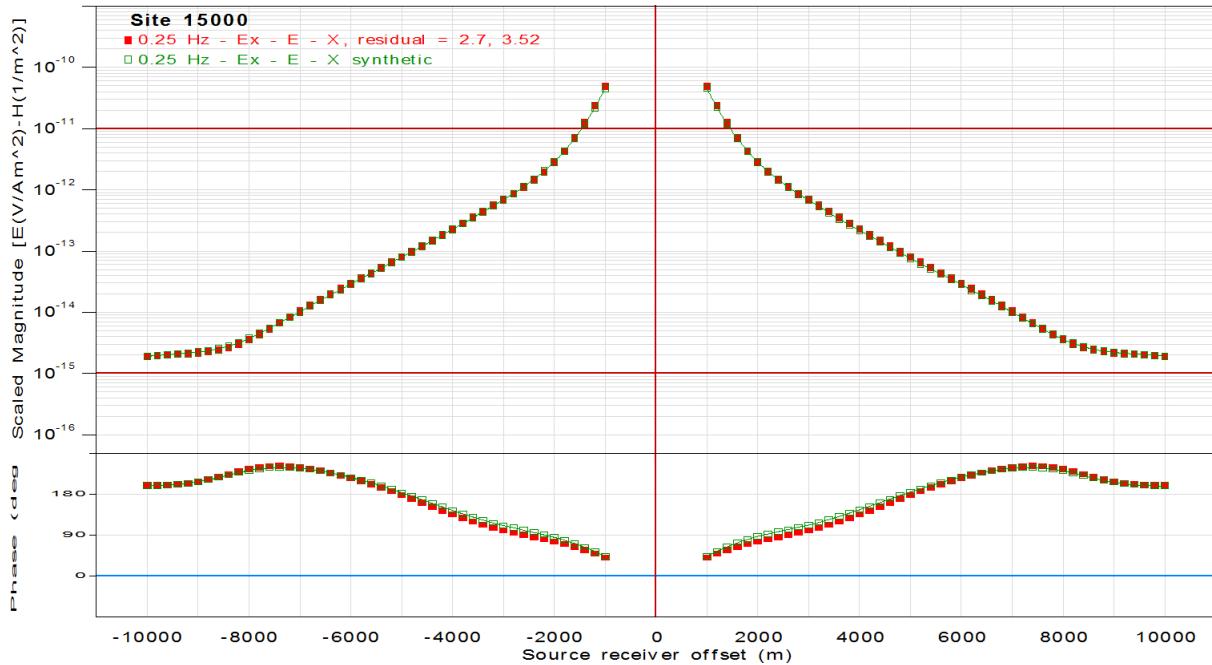


Figura 25 – Erro residual no sítio 16000

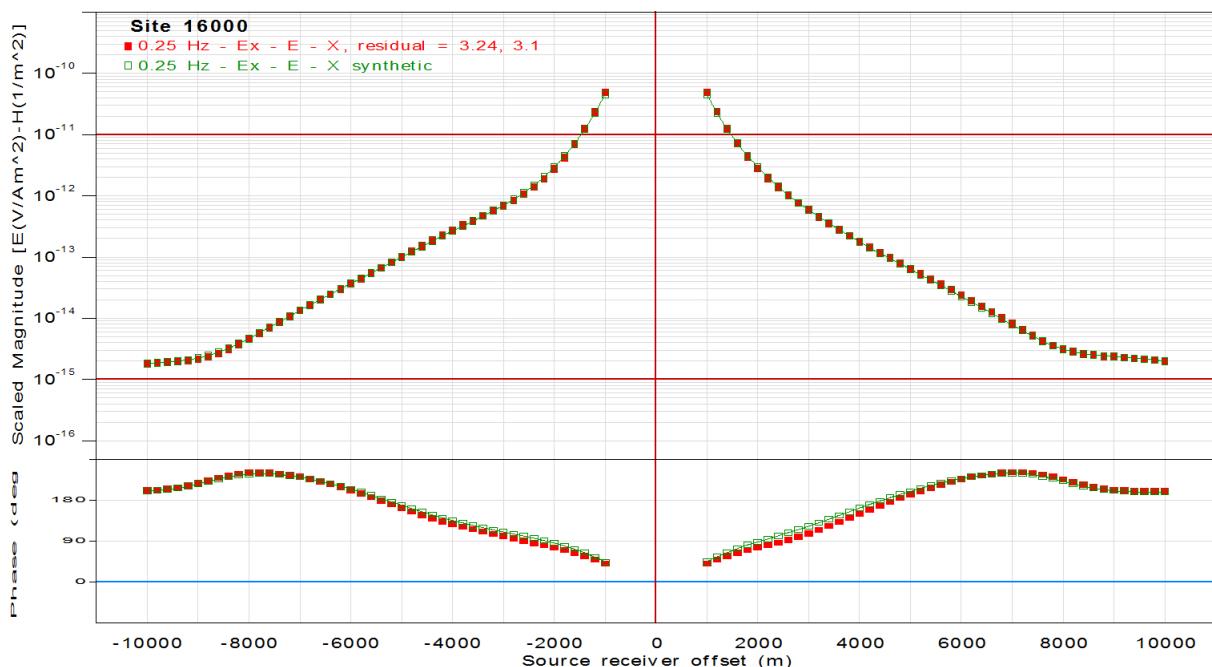


Figura 26 – Erro residual no sítio 17000

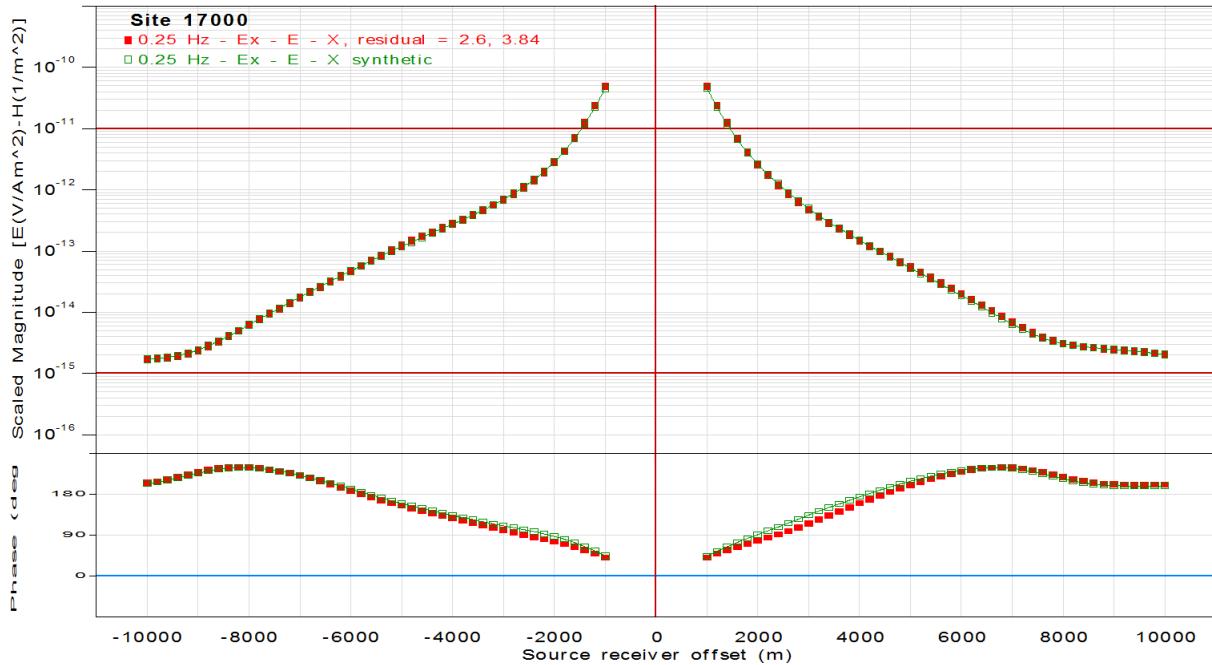


Figura 27 – Erro residual no sítio 18000

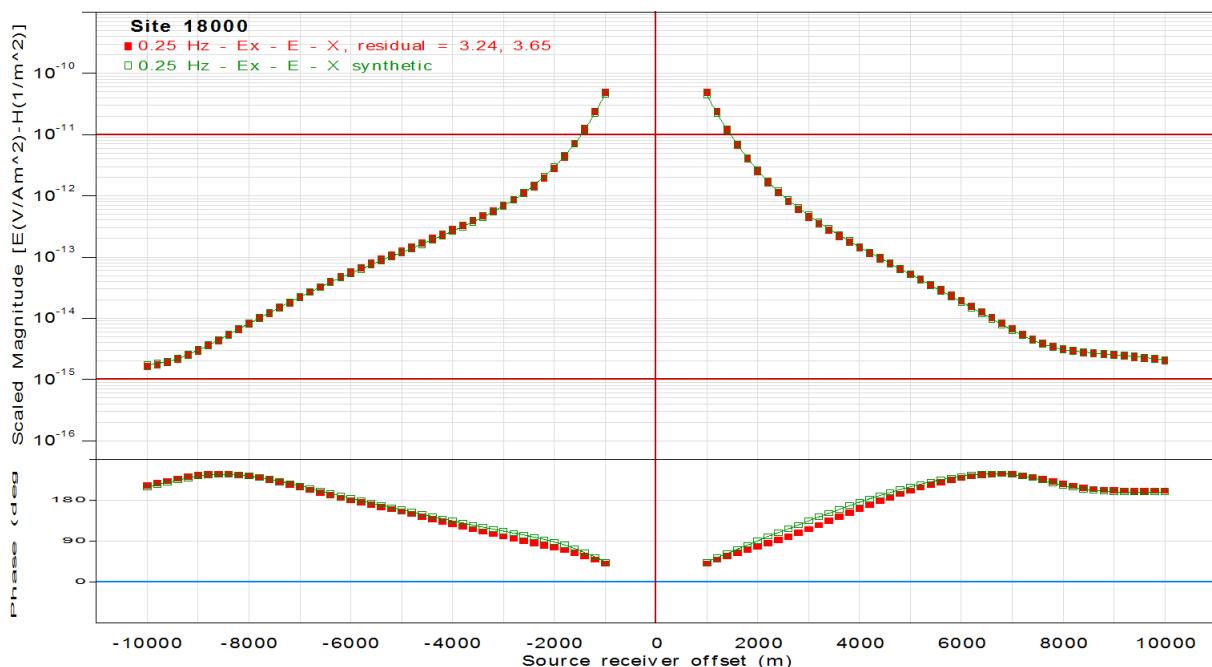


Figura 28 – Erro residual no sítio 19000

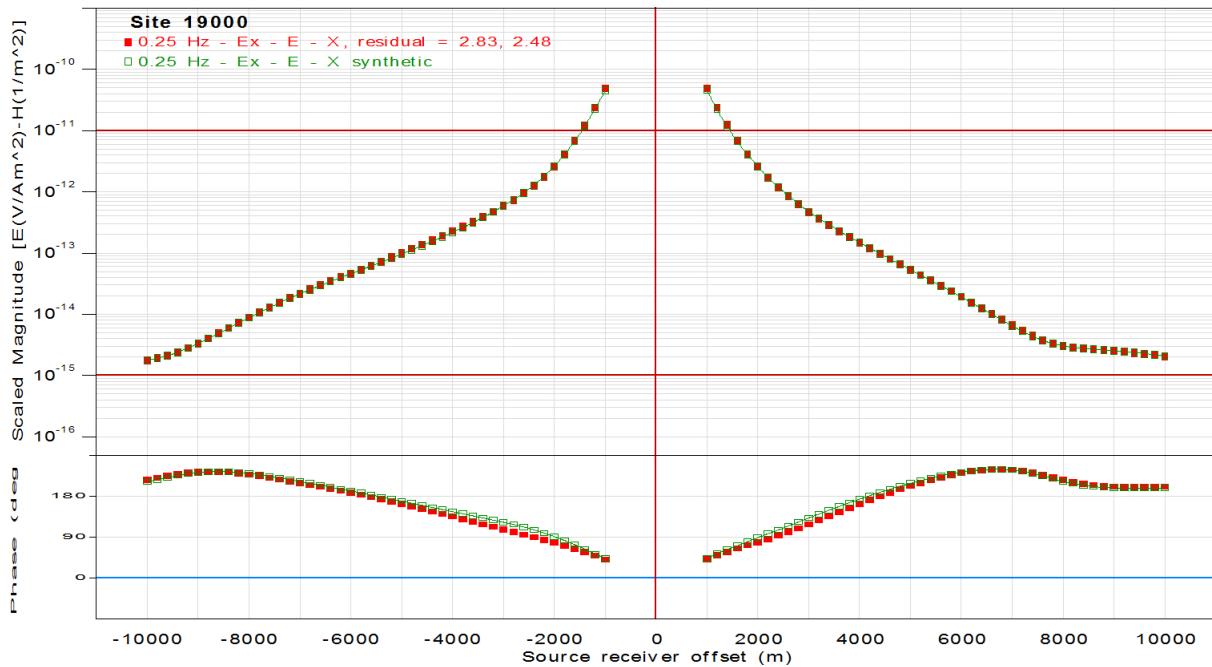


Figura 29 – Erro residual no sítio 20000

