地球ダイナモの新しいシミュレーションコード開発とその応用

山浦 優気

神戸大学 工学部

陰山 聡

神戸大学 システム情報学研究科

１．はじめに

地球磁場の起源を探るための地球ダイナモシミュレーションについて報告する。このシミュレーションの地球科学的背景や、目的、シミュレーション手法の詳細については、既に過去の2回の報告［1,2］で詳述しているので、ここではごく簡単に紹介する。

地球磁場の源は地球内部の外核と呼ばれる層にある液体鉄である。この外核の液体鉄が対流運動するために、その運動エネルギーが MHD（Magnetohydrodynamics）ダイナモ作用によって磁場のエネルギーに変換されている。

シミュレーションモデルとして、地球の外核を想定し、二つの同心球面に挟まれた球殻状の領域を考える。その中に電気伝導性流体（MHD 流体）が入っている。内側の球面（半径 $r = r_i$）は高温、外側の球面（半径 $r = r_o$）は低温に保たれている。球の中心方向に重力がはたらき、二つの球殻は同じ角速度 Ω で回転する。温度差が十分に大きければ（レイリー数 Ra が十分高ければ）内部の流体は熱対流運動し、MHD ダイナモ機構によって、対流の運動エネルギーが磁場のエネルギーに変換され、磁場が生成される。

この問題の基本方程式は MHD 方程式である。インヤン格子[3]と名付けた球面格子を用いて MHD 離散化する。空間方向には 2 次精度差分法を時間方向の積分には 4 次精度 Runge=Kutta 法を用いる。このコードは地球シミュレータ向けに最適化しており、2004 年のゴードン・ベル賞（15.2 TFLOPS）を受賞したものである［4]。その高速計算速度を活かした世界最高解像度での地球ダイナモシミュレーションを通じて、外核中の新しい電流構造と対流構造の発見[5]や、帯状の流れ構造の発見[6]などの成果を地球シミュレータで挙げることができた。今回のこの報告では、このコードを HA8000 向けに最適化するために行ったいくつかの準備的な研究の報告を行う。

２．コンパイルオプションによる最適化

以下では、テストに用いた計算の規模はインヤン格子の総格子点数（＝Nr x Nt x Np x 2）で示す。ここで、Nr は半径方向の格子点数、Nt は緯度方向 90 度の格子点数、Np は経度方向270 度の格子点数である。最後の x 2　は、イン（Yin）格子とヤン（Yang）格子である。

まずはじめに、各コンパイラによる性能評価を行った。使用したコンパイラは、HA8000 クラスタシステムから提供されている日立製 Fortran コンパイラ、Intel Fortran Compiler 11.0、PGI Fortran コンパイラ、gFortran の 4 種である。プログラムはどのコンパイラでもコンパイルでき、正常に実行することができた。

性能評価をノード数 4 で 1 ノードあたり 16 コアの flat MPI で、総格子点数を 824408 として

行った。時間次元のステップ数を 100 以上になるように約 5 分間ジョブを流したが、PGI のコンパイラ、gFortran についてはステップ数が 100 に達しなかったため、プログラムが正常に終了せず、性能評価ができなかった。

各コンパイラにおける実行結果を表 1 に記す。日立製コンパイラに比べ、Intel 製コンパイラが 20%ほど早かった。

コンパイラ	オプション	GFlops
日立製 Fortran コンパイラ	-O3 -noparallel	13.01
	-Oss -noparallel -autoinlice=2 -nolimit -noscope	17.04
	-O4	15.15
Intel Fortran Compiler 11.0	-O3 -xSSE3 -msse3	20.66
	-O3 -xSSE3 -msse3 -ipo	20.63
	-O3 -xSSE3 -msse3 -ipo9	20.69
	-O3 -xSSE3 -msse3 -ipo20	20.62
	-O4	20.7
PGI Fortran コンパイラ	-fast -O3 -tp=barcelona-64	
gFortran	-O3 -m64 -march=opteron	

表 1 ：コンパイラごとの性能評価

また、8 ノード 1 ノードあたり 16 コアの flat MPI、総格子点数を 50658300 として、4 ノードで性能の良かったオプションで日立製 Fortran コンパイラ、Intel Fortran Compiler それぞれ実行してみた。その結果を図 1 に示す。

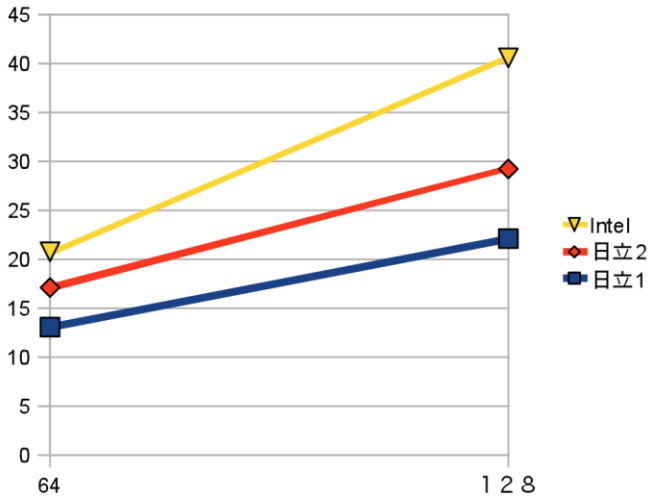

図 1 ：4 ノード使用時と 8 ノード使用時の速度（GFLOPS）

縦軸は GFlops、横軸はコア数である。日立 1 はオプション-O3 -noparallel、日立 2 はオプション-Oss

-noparallel -autoinline=2 -nolimit -noscope、Intel はオプション-O3 -xSSE3 -msse3 -ipo9 である

図１が示すとおり、Intel 製コンパイラは 128 ノードでも良い性能を出している。4 ノードでは 20%だった差が、8 ノードでは 50%まで増えている。

３．gprof によるプロファイリング

陰山が作成したオリジナルのシミュレーションコードでは、Fortran90 の system_clock 関数を使った自作の timer module による計時機能が組み込まれていた。この timer module を用いてプログラム実行時は常にある程度のプロファイリングを行うようにしているが、通信部分と計算部分を区別したより詳細なプロファイリングを行うためにプロファイリングツールである gprof を使用した。

gprof は、あるルーチンの実行にかかった時間を計測することが可能である。また、そのルーチンに含まれる外部関数の処理にかかった時間を計測することにより、その関数内部の計算だけでどれだけの時間がかかったか知ることができる。コンパイラは gprof と相性の良い Intel Fortran Compiler を使用し、オプションは-pg -O3 を利用した。8 ノード 1 ノードあたり 16 コアの flat MPI で実行した、その結果が図２である。

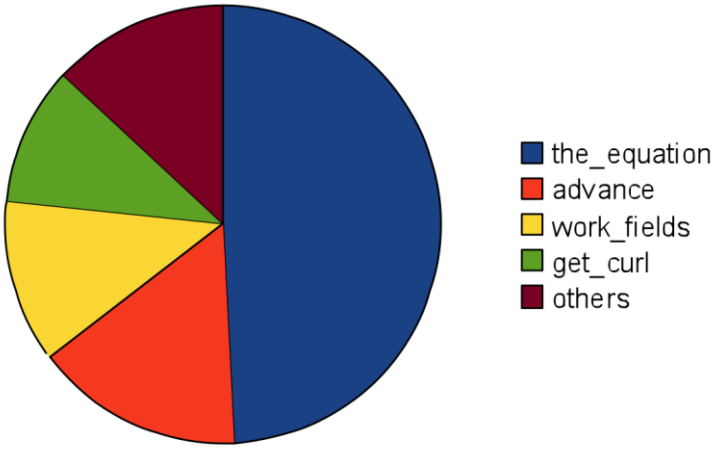

図２：gprof によるプロファイリング結果

the_equation は差分化された MHD 方程式を計算する部分、advance はルンゲ・クッタ法による積分ルーチン、work_fields は MHD の基本場以外の補助的な 3 次元場を基本場から計算するルーチン、get_curl はベクトル場の curl を計算するルーチンである。others と示した部分以外は、それぞれのルーチン内での処理のみにかかった時間である。すべてループ構造の計算を含み、MPI 通信や MPI_Barrier 等の関数、シミュレーションの初期化等は全て others に含まれる。

図２から、 MHD 差分式の計算部分にほぼ 50%近くの時間がかかっており、それ以外の MHD ソルバ関係のルーチンにほぼ全ての計算時間が費やされており、極めて健全なシミュレーションコードになっていることがわかる。地球シミュレータ向けに最適化されたコードではあるが、その基本構造は HA8000 でも大きく変える必要はなさそうである。

４．キャッシュヒット率の計測

次に、このコードの最大のホットスポットである MHD のソルバー部分を解析した。このルーチンは、3 次元空間に対応した 3 重 do-lood である。このループのキャッシュヒット率をキャッシュシミュレータで計測した。キャッシュヒット率を計測するツールとしては、oprofile や cachegrind (Valgrind) などがあるが、HA8000 クラスタシステムでは、oprofile を使用できなかったため、cachegrind を使用した。

cachegrind はメモリ関連のプロファイリングツールである Valgrind に付属するプロファイラーであるが、I1、 L1、 L2（最新版ではもっとも大きなキャッシュである LL キャッシュ）の 3 つのキャッシュをシミュレートすることによって、キャッシュヒット率を擬似的ではあるが、計測することができる。ただし、あくまでシミュレータであることや、HA8000 クラスタシステムに搭載されている CPU である Opteron には L3 キャッシュ（しかもサイズ 2MByte と大きい）が搭載されているため、必ずしも正確なキャッシュヒット率とはならないことに留意する必要がある。キャッシュシミュレータによる実行結果を図３に示す。

なお、HA8000 クラスタシステムに cachegrind は、はじめからインストールされていたが、バージョンが古く、MPI 関連の最適化がなされていない事や複数プロセスのプロファイル結果を合成することができないため、新たにバージョン 3.5.0 をコンパイルして使用した。

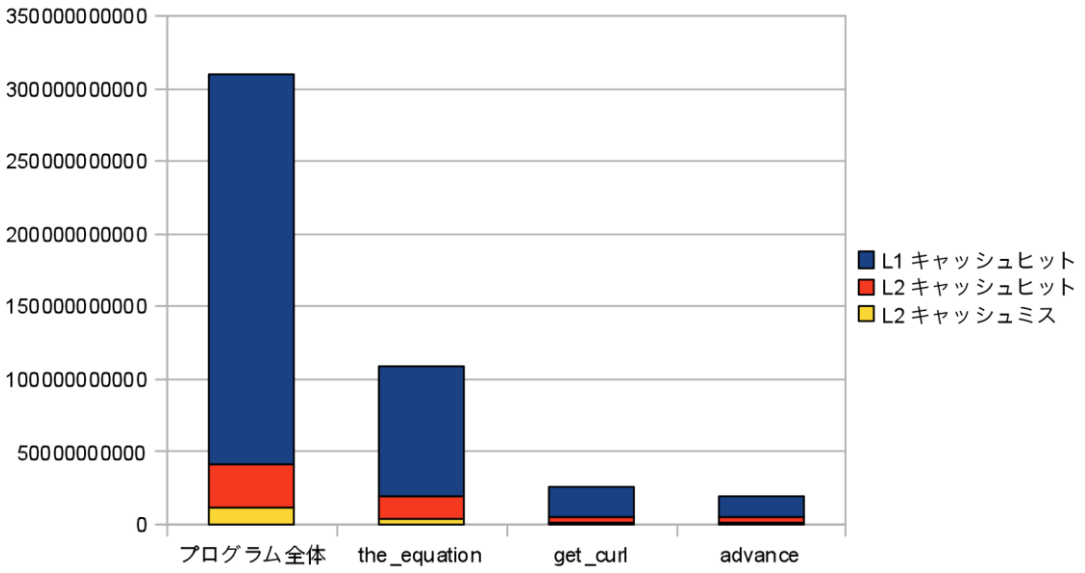

図３：cachegrind シミュレータによるキャッシュヒット率

青色が L1 キャッシュヒット数つまり L1 キャッシュから取り出せた結果で、赤色と黄色を合わせたものが L1 キャッシュミス数、黄色は L1 キャッシュ、L2 キャッシュから取り出せなかったものの数である。

５．ループ分割による最適化

ホットスポットを確認し、キャッシュヒット率を計測したところで、いくつか最適化を試みた。メイン計算部分は３重ループで最内ループ内に複雑に多くの計算式がつらなっているが、これをいくつかのループに分割することを試みた。

まず、ループ内で依存関係がない２つのブロック見つけ、単純にこれらを２つに分割してみた（方法１）。

MHD 方程式の右辺には、依存関係はあるものの、temporary な変数（例えば基本場の 2 次の非線形項）の設定部分と、それを利用する部分の二つに分けられるものが多い。そこで、そのような temporary な変数を１次配列に変更し、以下のように３重ループの最内ループのみを分割する方法も試してみた（方法２）。

変更前	変更後（方法２）

do k = 1 , NK
  do j = 1 , NJ
  do i = 1 , NI
  tmp = a1(i,j,k)*a2(i,j,k)
  b1 = tmp**2
  b2 = tmp*a3(i,j,k)
  end do
  end do
end do
	do k = 1 , NK
	  do j = 1 , NJ
	  do i = 1 , NI
	  tmp(i) = a1(i,j,k)*a2(i,j,k)
	  end do
	  do i = 1 , NI
	  b1 = tmp(i)**2
	  b2 = tmp(i)*a3(i,j,k)
	  end do
	  end do
	end do

MHD 計算は 8 つの物理場をもつが、上記の temporary 変数を１次配列にしたことで、各物理量の計算を分割することが可能になったため、それらの計算を 8 つに分割してみた。上記では、b1、b2 の計算部分を分割する。（方法３）

キャッシュシミュレータによる結果を図４、実行結果を表２に記す。いずれも 8 ノード 128 コアでの実行結果である。

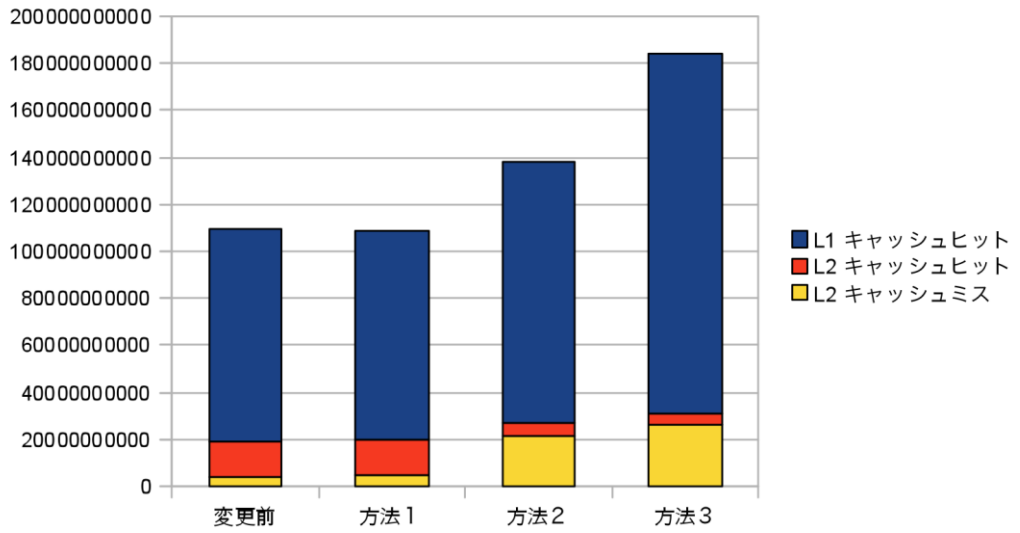

図４：ループ分割のキャッシュヒット

各手法の MHD ソルバー部分 1STEP のみの結果。

	GFlops
変更前	40.58
方法１	39.15
方法２	37.99
方法３	36.28

表２：ループ分割の Flops 値

200STEPS でのプログラム全体の結果。右の値は GFlops 値。方法２、方法３では明らかに遅くなっている。

方法２、方法３では、図４のようにキャッシュ読み込み数が大幅に増え、L2 キャッシュミスヒット数も増えており、表２から分かるようにプログラム全体が遅くなる結果となった。いずれも、性能をあげることはできず、むしろ遅くなってしまう場合もあった。

６．行列の入れ替え

次に行と列を入れ替える手法を試みた。これは以下の「変更前」のプログラムのように、b のアクセスが不連続なため､キャッシュミスが多発するのを事前に b2 という転置行列を作成し防ぐものである。

変更前	変更後（方法４）

do j = 1 , NJ
  do i = 1 , NI
  a(i,j) = a(i,j) + b(j,i)
  end do
end do
	do i = 1 , NI
	  do j = 1 , NJ
	  b2(i,j) = b(j,i)
	  end do
	end do
	
	do j = 1 , NJ
	  do i = 1 , NI
	  a(i,j) = a(i,j) + b2(i,j)
	  end do
	end do

MHD ソルバー部分にこのような配列があったため、この方法を試みたのが（方法４）である。

また、いくつかあるループ内において、該当の配列へのアクセスが連続となっているのが、主にプログラムの初期化部分であったため、配列を元から行と列を入れ替えたのが（方法５）である。これらの結果を図５、表３に示す。

なお、これらの図表における、「変更前」のキャッシュヒット率や GFlops 値はこれまでに記述したチューニングを施した上での実行である。

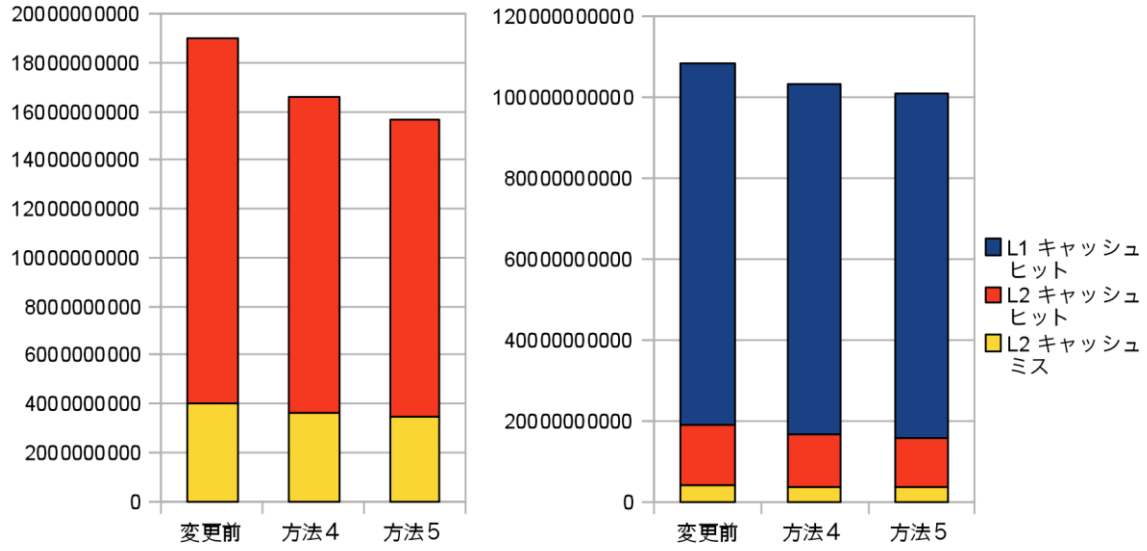

図５：行列の入れ替えによるキャッシュヒット率の変化

各手法の MHD ソルバー部分 1STEP のみの結果。8 ノード 128 コアで実行した。

	GFlops
変更前	40.85
方法４	41.78
方法５	43.66

表３：各手法（行列の入れ替え）の GFlops 値

200STEPS でのプログラム全体での結果。8 ノード 128 コアで実行した。

図５は MHD ソルバー部分のみの結果である。方法４と方法５ではそれほど差が見られないものの、方法５はプログラム全体に渡って変更したため、MHD ソルバー部分以外でのキャッシュミスヒット数が減り、その結果として表３のように全体で 7%ほどの高速化になった。方法４、方法５共に、キャッシュミスヒット数が減った。

７．その他の最適化

非常に単純ではあるが、ある方法によりキャッシュミスが減ったので記しておく。以下のプログラムにおいて、配列 a へのアクセスを明示的に変更する事によって若干ではあるがキャッシュミスが減少した。これは、コンパイラが自動で行ってくれていると判断していたが、３重ループ構造が複雑であるためか、意外にもそうではなかったようである。この結果は図９のようになり。1%ほどではあるが高速化された（表４）。

変更前	変更後

do j = 1 , NJ
  do i = 1 , NI
  b(i,j) = a(i+1,j) + a(i-1,j)
  end do
end do
	do j = 1 , NJ
	  do i = 1 , NI
	  b(i,j) = a(i-1,j) + a(i+1,j)
	  end do
	end do

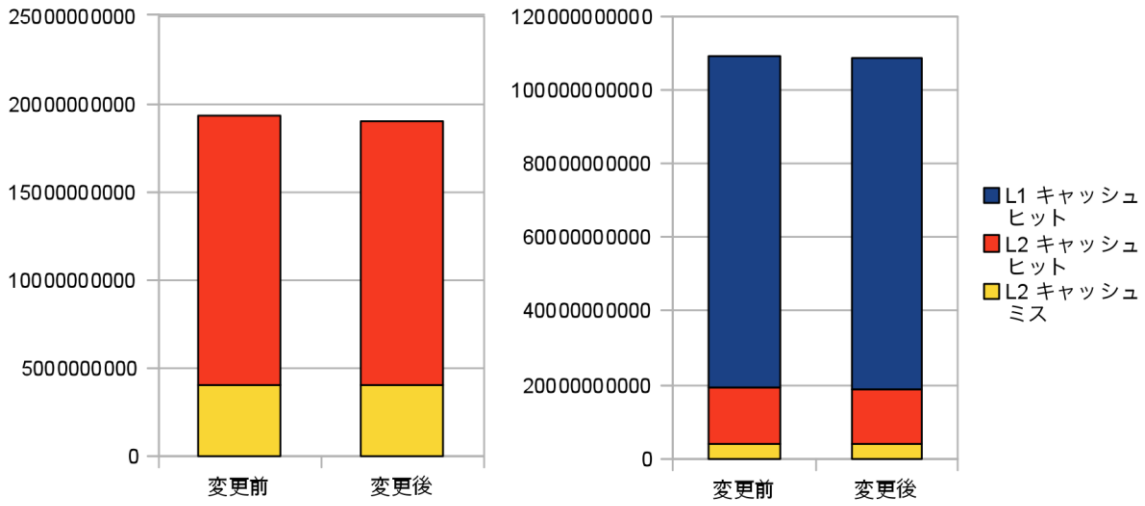

図６：項の順番入替による効果

各手法の MHD ソルバー部分 1STEP のみの結果。8 ノード 128 コアで実行した。

	GFlops
変更前	40.58
変更後	40.85

表４：項の順番入れ替えによる GFlops 値の変化

若干ではあるが、L1 キャッシュミス数、L2 キャッシュミス数が減っている。

８．まとめ

地磁気ダイナモ MHD シミュレーションコードのチューニングを目指し、準備的な研究を進めた。今回は主に cachegrind というキャッシュシミュレータを使い、性能評価しつついくつかの最適化を試みた。演算性能の向上という点では大きな結果は得られなかったが、今後チューニングを進めるにあたり指針となる結果が得られた。ただし、cachegrind はあくまでシミュレータであることと、実行時のオーバーヘッドが大きく、数ループ分のデータしかとれなかったのが残念である。実行時オーバーヘッドの少ない oprofile もしくは他のサンプリング型キャッシュプロファイラーを使用できるようになれば大変ありがたいと感じた。

今回のテストはあくまで準備的なものである。今後、より本格的なキャッシュチューニングや、その他のチューニング技法を活用し、さらなる最適化を進めたい。

参 考 文 献

[1] 陰山聡、大野暢亮、HA8000 システムでの地球ダイナモシミュレーションと可視化、スーパーコンピューティングニュース, Vol.11, No. Special Issue 2, pp.33-42 (2009)

[2] 陰山聡、大野暢亮、地球ダイナモの新しいシミュレーションコード開発とその応用、スーパーコンピューティングニュース, Vol.12, No. Special Issue 1, pp.57-62 (2010)

[3] Akira Kageyama and Tetsuya Sato, “Yin-Yang Grid”: An Overset Grid in Spherical Geometry. Geochem. Geophys. Geosyst. vol.5, doi:10.1029/2004GC000734 (2004)

[4] A. Kageyama et al., A 15.2 TFlops Simulation of Geodynamo on the Earth Simulator, Proceedings of the ACM/IEEE SC2004 Conference, pp.35-43, 2004

[5] A. Kageyama, T. Miyagoshi, and T. Sato, Formation of current coils in geodynamo simulations, Nature, vol.454, pp.1106-1109 (2009)

[6] T. Miyagoshi, A. Kageyama, and T. Sato, Zonal flow formation in the Earth's core, Nature vol.463, pp.793-796 (2010)