計算機アーキテクチャ 142−16
ハイパフォーマンス 85−16
コンピューティング
（2001. 3. 8）

キャッシュラインを考慮した3次元PDE solverの最適化手法

近藤正章† 岩本貢† 中村宏†

近年プロセッサと主記憶の性能格差の問題が深刻化している。そのため、キャッシュブロッキングなどの手法により、データアクセスの局所性をできる限り向上させ、主記憶へのアクセスを低減することが高性能を得るために重要となる。本稿では、HPC分野において重要性が増しつつある3次元PDE solverについて、キャッシュラインを考慮し、主記憶アクセス時のデータトラフィックを少く抑えることができるブロックサイズ選択法を提案する。また、既存の手法と本手法との性能比較を行い、提案する手法の有効性を示す。提案する手法は既存の手法に比べ、キャッシュミス回数を削減することができ、高性能が得られることがわかった。

Cache Line Impact on 3D PDE Solvers

MASAAKI KONDO,† MITSUGU IWAMOTO† and HIROSHI NAKAMURA†

Because performance disparity between processor and main memory is serious, it is necessary to reduce off-chip memory accesses by exploiting temporal locality. Loop tiling is a well-known optimization which enhance data locality. In this paper, we show a new cost model to select the best tile size in 3D partial differential equations (PDEs) solvers. Our cost model carefully takes account of cache line impact. Thus, it successfully reduces data traffics between cache and main memory or lower level cache. We also present performance evaluation of our cost model. The evaluation results reveal the superiority of our cost model to other cost model proposed so far.

1. はじめに

近年プロセッサと主記憶の性能格差の問題が深刻化している。そのため、データアクセスの局所性を利用して、主記憶へのアクセスを低減することが必須である。キャッシュブロッキング(タイリング)[1]は、ソフトウェア的にデータアクセスの時間的局所性を向上させる手法であり、主記憶に比べ高速なキャッシュを最大限に活用することは、高性能を得るために重要となる。

一方、大規模科学技術計算などに代表される、ハイパフォーマンスコンピューティング(HPC)分野において、偏微分方程式(PDE)を高速に解くことは非常に重要である。SPEC[2]やNPB[3]といった著名なベンチマークにも、偏微分方程式の解を有限差分法により求めるカーネルルーチン(PDE solver)が含まれており、その重要性が伺える。

歴史的にPDE solverは2次元空間上の問題がターゲットとされてきた。そのため、2次元PDE solverにおけるキャッシュ最適化手法はこれまでも数多く提案されている。しかし、近年では、プロセッサの処理能力向上により3次元空間上のPDEを解くことが広く行われるようになっている。3次元のPDEを解く場合、2次元の問題に比べ、時間的局所性のあるデータへのアクセスの時間間隔が長くなることから、メモリアクセスがボトルネックとなることによる性能低下が著しい。そこで、3次元PDE solverの性能最適化手法を検討することは特に重要であると考えられる。

Riveraらは3次元PDE solverに関して、キャッシュブロッキング、およびpadding手法を提案している[4]。Riveraらの手法は、最適なブロックサイズを決めるために“コスト関数”と呼ばれる式を用いる。このコスト関数を用いることで、プログラマやコンパイラは適切なブロックサイズを一意に決定することができる。しかし、彼らのコスト関数にはキャッシュラインの影響が考慮されていないという問題があった。

そこで本稿では、キャッシュラインの影響を考慮した新しいコスト関数を提案する。提案するコスト関数を用いることで、Riveraらの提案するコスト関数に比べ、主記憶とキャッシュ間のデータトラフィックを削減することができ、さらなる高性能が期待できる。本稿では、両コスト関数を用いた性能比較を行う。

次節では、文献[4]で提案されている3次元PDE

† 東京大学 先端科学技術研究センター
Research Center for Advanced Science and Technology, The University of Tokyo

```
A(N,N), B(N,N)
do J=2, N-1
  do I=2, N-1
    A(I,J) = C * (B(I-1,J) + B(I+1,J) +
                  B(I,J-1) + B(I,J-1) )
```

図1　2次元Jacobiカーネルコード

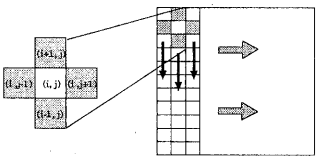

図2　2次元Jacobiカーネルアクセスパターン

solverのキャッシュブロッキング手法や、コスト関数について簡単に述べる。3節では、キャッシュラインの影響を考慮した新たなコスト関数を提案する。4節ではいくつかの3次元PDE solverのカーネルルーチンを用いての性能評価を行い、5節でまとめと今後の課題について述べる。

2. 3次元PDE solverのブロッキング手法

本節では、文献[4]で述べられている、3次元PDE solverにおけるブロッキング手法について簡単に紹介する。

2.1 3次元PDE solverの概要

まず、PDE solverのアクセスパターンの概要を示すため、2次元のJacobiカーネルを例として説明する。図1に、2次元Jacobiカーネルのコードを、図2にそのデータアクセスパターンを示す。

2次元Jacobiカーネルでは、格子上のある1点(図1の$A(I,J)$)を計算するために、周囲の4点(図1の$B(I\pm1,J\pm1)$)をアクセスする。ループが繰り返される度に、2次元配列上の各格子点についてそのアクセスが行われる。このように、決まった型(*stencil*)のアクセスが、配列の各格子点に対し繰り返されることから、このような計算を「stencil計算」と呼ぶ。

2次元Jacobiカーネルの場合、図2の3列がキャッシュに収まるサイズであれば、配列Bの再利用性を最大限に活用できる。すなわち、1列のサイズをNとすると、キャッシュサイズが$3N$以上であれば、一度$B(I,J+1)$のアクセスでキャッシュに転送されたデータは、stencilの他の3点としてアクセスされる際にもキャッシュ上にあることが予想される⋆。従って、配列Bの再利用性が完全に活用できることになる。

次に、3次元のstencil計算を考える。図3に、3次元Jacobiカーネルのコードを、図4にそのデータアクセスパターンを示す。3次元Jacobiカーネルでは、格子上のある1点(図3の$A(I,J,K)$)の値を求めるために、周囲の6点($B(I\pm1,J\pm1,K\pm1)$)をアクセスする。このように、3次元Jacobiカーネルでは3枚のI-J平面($(K-1)$、K、$(K+1)$のI-J平面)を同時にアクセスするため、stencilアクセス6回分にわたる再利用性を全て活用するためには、$3N^2$以上ものキャッシュサイズが必要となる⋆⋆。

```
A(N,N,N), B(N,N,N)
do K=2, N-1
 do J=2, N-1
  do I=2, N-1
   A(I,J,K) = C * (B(I-1,J,K) + B(I+1,J,K) +
                   B(I,J-1,K) + B(I,J+1,K) +
                   B(I,J,K-1) + B(I,J,K+1) )
```

図3　3次元Jacobi Kernelコード

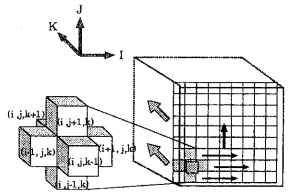

図4　3次元Jacobi Kernelアクセスパターン

16KB(double precision型データ2048要素)のキャッシュを仮定した場合、2次元Jacobiカーネルでは、配列サイズが$1024\times M$ (Mは任意の数)以下であれば再利用性を最大限に活用できる。一方、3次元Jacobiカーネルでは配列サイズが$32\times32\times M$以下でないと、再利用性を活用することができない。数MBものキャッシュを持つプロセッサにおいても現実の問題サイズを考えると、再利用性を活用することは難しいと予想される。そのため、特に3次元のstencil計算ではキャッシュブロッキングが重要となる。

2.2 ブロッキング手法

3次元Jacobiカーネルにおいて、配列Bの再利用性を最大限に活用するためには、キャッシュ上に3次元格子空間上の3つのI-J平面が載る必要がある。従って、3つのI-J平面がキャッシュサイズに収まるようにI-J平面を分割することでキャッシュブロッキングを行う。これは、図3のI,Jループをブロック化することで実現できる。図5に3次元Jacobiカーネルのコードを上記の手法でブロッキングしたコードを、図6にそのアクセスパターンの概要を示す。なお、このブロッキング手法は、他のstencil計算においても同様である。

2.3 コスト関数

3次元stencil計算にキャッシュブロッキングを適応する際、ブロックサイズ(図6のTI、およびTJ)を決める必要がある。stencil計算のブロッキングでは、ブロック内のデータのみではなくブロックの境界に位置するデータも同時にアクセスされる。この境界上のデータはブロッキングを行っても再利用性が向上しないため、できる限り境界に位置するデータの数が少なくな

⋆ 厳密にはキャッシュサイズが$2N+1$以上であればよい

⋆⋆ 厳密にはキャッシュサイズが$2N^2+1$以上であればよい

表1　コンフリクトしないブロックサイズの例 (問題サイズ: $200 \times 200 \times M$)

TK	1	1	1	1	2	2	2	2	3	3	3	4	4	4	...
TJ	1	10	41	256	1	4	5	15	5	11	15	4	15	56	...
TI	2048	200	48	8	960	200	160	40	72	40	24	72	16	8	...

```
do JJ=2, N-1, TJ
 do II=2, N-1, TI
  do K=2, N-1
   do J=JJ, min(JJ+TJ-1, N-1)
    do I=II, min(II+TI-1, N-1)
     A(I,J,K) = C * (B(I-1,J,K) + B(I+1,J,K) +
                     B(I,J-1,K) + B(I,J+1,K) +
                     B(I,J,K-1) + B(I,J,K+1) )
```

図5　3次元Jacobi Kernelコード

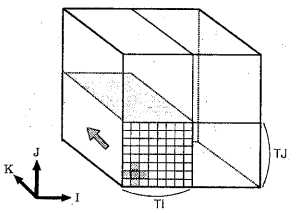

図6　3次元Jacobi Kernelアクセスパターン

るようにブロッキングすることが望ましい。Riveraらは文献[4]において、上記の事柄を式で表した"コスト関数"を提案している。以下にそのコスト関数を示す(コスト関数の導出については文献[4]を参照)。

$$Cost_{Riv} = \frac{(TI+2)(TJ+2)}{TI \times TJ} \qquad (1)$$

このコスト関数は、stencil計算にキャッシュブロッキングを適応した際、境界に位置するデータの要素数に比例する。そのため、このコスト関数ができる限り小さい値をとるようなブロックサイズ$TI \times TJ$が、最適なブロックサイズである。$TI \times TJ$を一定とすると、上記のコスト関数はTIとTJが等しいときに最小となるため、Riveraらはできるだけ正方形に近いブロックサイズが良いと結論付けている。このことは、面積が同じである四角形の辺(境界)の長さの合計は、その四角形が正方形のときに最小となるという概念と一致する。

2.4 Avoiding Conflicts

キャッシュブロッキングを行っても、キャッシュ上でラインコンフリクトが生じると性能が著しく低下する[1]。ラインコンフリクトを防ぐために、これまで多くの手法が提案されている[5]。

Riveraらは3次元のstencil計算において、コンフリクトを防ぐために以下の3つのアルゴリズムを提案している[4]。なお、これらのアルゴリズムの詳細については文献[4]を参照されたい。

- Euc3D: 文献[6]で述べられている、Euclideanアルゴリズムを3次元に拡張したアルゴリズム。
- GcdPad: キャッシュサイズに収まるような、最大の2の巾乗のブロックサイズを求め、コンフリクトを防ぐためにpaddingを施す。
- Pad: 上記のEuc3DとGcdPadを複合したアルゴリズム。

例えば、上記のEuc3Dアルゴリズムを用いることで、コンフリクトしないブロックサイズの候補がいくつか得られる。配列サイズ$200 \times 200 \times M$、キャッシュサイズ16KBの条件の基でEuc3Dのアルゴリズムを適用した場合、表1に示すような、ラインコンフリクトが起きないブロックサイズの候補が得られる。前節で述べたコスト関数は、この候補の中から最適なタイルサイズを選ぶ目的で用いられる。表1にコスト関数$Cost_{Riv}$を適用すると、$(TI, TJ) = (24, 15)$の時にコスト関数が最小となる。従って、$(TI, TJ) = (24, 15)$というブロックサイズが最適なブロックサイズとして用いられる。

3. コスト関数の提案

Riveraらが提案するコスト関数$Cost_{Riv}$は非常に簡単なものであり、キャッシュラインの影響を考慮していなかった。しかし、キャッシュラインの影響を考慮すると、必ずしも正方形に近いブロックが最適であるとは限らない。本節では、キャッシュラインの影響を考慮した新たなコスト関数を提案する。

3.1 キャッシュトラフィック

3次元のstencil計算を行う際に、キャッシュに転送される要素数をキャッシュトラフィックと呼ぶことにする。無限容量のキャッシュサイズを仮定すると、配列サイズN^3の場合では、キャッシュトラフィックはN^3となる。しかし、限られたキャッシュ容量のもとでキャッシュブロッキングを行うと、ブロックの境界に位置するデータは再利用されず、同じデータが複数回に渡ってキャッシュに転送されることになる。従って、キャッシュブロッキングを行った場合でも、キャッシュトラフィックはN^3より増えてしまう。

そこで、このキャッシュトラフィックの増加という視点からコスト関数を議論する。$Cost_{Riv}$では、このトラフィックの増加は、I-J平面のブロックの境界に位置する2要素ずつであるとを示唆している。I-J平面において、J方向を分割した場合にはこれは正しい。しかし、I方向を分割した場合、トラッフィクの増分は2要素ではなくラインサイズ(L)分であると考えられる。

図7にブロッキングを適応したI-J平面の様子を示す。図中の*boundary 1-2*とは、ブロック1を計算する

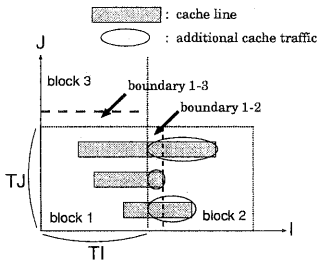

図7　ブロック1の計算

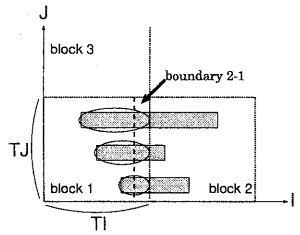

図8　ブロック2の計算

際に必要となるブロック2に含まれるデータを意味する。例えば、図7でブロック1を計算する時には、ブロック1内のデータのみではなく、*boundary 1-2*、および*boundary 1-3*のデータも必要になる。ここで、キャッシュミスが生じた際にはキャッシュライン単位でデータが転送されることをふまえると、I方向に分割した場合にブロック1以外で転送されるデータは、境界の*boundary 1-2*だけでなく、図7中に丸で囲まれたデータとなる。これが実際のキャッシュトラフィックの増加分となる。

次に、ブロック2を計算する場合(図8)を考える。前節で述べたブロッキング手法では、ブロック1のI-J平面をK方向にわたってアクセスするため、ブロック2に計算が移った時、該当するブロック1のI-J平面のデータはすでにキャッシュから追い出されている。そのため、図8の*boundary 2-1*に位置するデータは再びキャッシュに転送される。ここでもブロック1の計算時と同様にキャッシュトラフィックの増加が発生する。ブロック1を計算する際に発生するトラフィックの増加と合計すると、I方向に分割した場合の各境界においては、最低でもキャッシュラインサイズL要素分がキャッシュトラフィックの増加となる。なお、I方向について、ブロックの境界とラインの端が一致すると、$2L$要素のトラフィック増加が生じることになる。

3.2　コスト関数

前節で述べたように、I-J平面をI方向に分割した場合には、各境界においておよそL要素のトラッフィクの増加が生じる。これを、コスト関数で表すことを考える。

mをI方向の分割数、nをJ方向の分割数とする。問題サイズN^3を仮定すると、$m = \lceil N/TI \rceil$、$n = \lceil N/TJ \rceil$となる。I方向に分割した場合、$L \times (m-1)$要素のキャッシュトラフィックの増加が生じ、J方向に分割した場合は$2 \times (n-1)$のトラフィック増加が生じる。従って、$N^2(L(m-1)+2(n-1))$要素が、問題サイズN^3に加えてキャッシュに転送される。キャッシュに転送される要素数の合計を見積もると、$N^3 + N^2((m-1)L + 2(n-1))$となる。ここで$N^3$、および$N^2$はブロックサイズ$TI$、$TJ$によらず一定であるので、定数項を削除して以下のコスト関数が得られる。

$$Cost_{New*} = L(m-1) + 2(n-1) \quad (m = \lceil \frac{N}{TI} \rceil, n = \lceil \frac{N}{TJ} \rceil) \qquad (2)$$

ブロックサイズ$TI \times TJ$を一定とすると、上記のコスト関数は$TI : TJ = L : 2$のときに最小となる。

次に、store先の配列(図5の配列Aにあたる)についても、同様にキャッシュラインの影響を検討する。キャッシュがwrite aroundキャッシュであった場合は、配列Aの要素はキャッシュに転送されない。しかし、write allocateキャッシュであった場合には、Aの要素もキャッシュに転送される。

配列Aは、stencilアクセスは行われないため、J方向に分割した場合、トラフィック増加は生じない。しかし、I方向に分割した場合は同様に、$L \times (m-1)$要素の余分なトラフィックが生じる可能性がある*。

さらに、stencil計算のカーネル中に他のアクセスされる配列が含まれている場合も同様に、$L \times (m-1)$の余分なトラフィックが発生する。これは後で示すRESIDカーネル(図10参照)に見られる。これらの配列の影響を$Cost_{New*}$に含めることで、提案するコスト関数を一般化する。

$$Cost_{New} = P \times L(m-1) + Q \times 2(n-1) \qquad (3)$$

ここで、Pはキャッシュに転送される配列の個数であり、Qはその中でstencilアクセスが行われる配列の個数となる。Jacobiカーネルを例にとると、Jacobiカーネルでは配列AとBがアクセスされるが、write aroundキャッシュの場合には、Bのみがキャッシュに転送されるため、Pは1となる。一方、write allocateキャッシュの場合は、配列A、Bともにキャッシュに転送されるため、Pは2となる。また、stencilアクセスされる配列はBのみであり、Qはキャッシュのwrite-miss ポリシーに関係なく1となる。

Jacobiカーネルにおいて、32Bキャッシュライン、write allocateキャッシュを仮定した場合($L=4, P=2, Q=1$)、表1に$Cost_{New}$を適用すると、最適なブロックサイズは$(TI, TJ) = (40, 11)$となる。

* TIがキャッシュのラインサイズの倍数であり、配列Aの先頭がラインサイズLにalignされている場合は、余分なトラフィックは生じない

```
do odd=0, 1
 do k=2, N-1
  do j=2, N-1
   do i=2+mod(k+j+odd,2), N-1, 2
     A(i,j,k) = C1 * A(i,j,k) +
                C2 * (A(i-1,j,k) + A(i,j-1,k) +
                      A(i+1,j,k) + A(i,j+1,k) +
                      A(i,j,k-1) + A(i,j,k+1) )
```

図9 Red-Black SORコード

```
do i3=2, dk-1
 do i2=2, N-1
  do i1=2, N-1
   r(i1,i2,i3) = v(i1,i2,i3)
     - A0 * ( u(i1,i2,i3) )
     - A1 * ( u(i1-1,i2  ,i3  ) + u(i1+1,i2  ,i3  )
            + u(i1  ,i2-1,i3  ) + u(i1  ,i2+1,i3  )
            + u(i1  ,i2  ,i3-1) + u(i1  ,i2  ,i3+1))
     - A2 * ( u(i1-1,i2-1,i3  ) + u(i1+1,i2-1,i3  )
            + u(i1-1,i2+1,i3  ) + u(i1+1,i2+1,i3  )
            + u(i1  ,i2-1,i3-1) + u(i1  ,i2+1,i3-1)
            + u(i1  ,i2-1,i3+1) + u(i1  ,i2+1,i3+1)
            + u(i1-1,i2  ,i3-1) + u(i1-1,i2  ,i3+1)
            + u(i1+1,i2  ,i3-1) + u(i1+1,i2  ,i3+1))
     - A3 * ( u(i1-1,i2-1,i3-1) + u(i1+1,i2-1,i3-1)
            + u(i1-1,i2+1,i3-1) + u(i1+1,i2+1,i3-1)
            + u(i1-1,i2-1,i3+1) + u(i1+1,i2-1,i3+1)
            + u(i1-1,i2+1,i3+1) + u(i1+1,i2+1,i3+1))
```

図10 RESIDカーネルコード

表2 評価に用いるマシン

	SUN Ultra30	SGI O2
CPU - Chip - Clock Cycle	 UltraSPARC-II 360MHz	 MIPS R10000 175 MHz
L1 Cache - Size - Line Size - write-miss policy	 16KB 16B (2 word/line) write around	 32KB 32B (4 word/line) write allocate
L2 Cache - Size - Line Size - write-miss policy	 4MB 64B (8 word/line) write allocate	 1MB 64B (8 word/line) write allocate

4. 性能評価

4.1 評価方法

Riveraらは文献[4]において、3つのカーネルルーチン(3D Jacobi、Red-Black SOR、RESID)を用いて性能評価を行っている。本稿でも同様のカーネルを用いて、Riveraらのコスト関数$Cost_{Riv}$と、我々のコスト関数$Cost_{New}$を比較評価する。図9にRed-Black SORカーネルを、図10にRESIDのカーネルルーチンを示す(ブロック化したコードは文献[4]を参照)。性能評価にはSun Ultra30および、SGI O2の2つのマシンを用いる。表2に各々のマシンの仕様を示す。

評価に用いる問題サイズは$N \times N \times 30$ (各データはdouble precision)とし、NはSun Ultra30では400〜600、SGI O2では200〜400と変化させ評価を行った。この問題サイズは、両マシンにおいて、問題サイズが小さい場合はいくつかの$N \times N$のI-J平面がL2キャッシュに載るが、問題サイズが大きくなるとL2キャッシュには収まりきらないように選んである。キャッシュブロッキングについては、L1キャッシュのみをターゲットとした。

表3 各マシン、各カーネルにおけるPとQの値

Platform	Program	for L1		for L2	
		P	Q	P	Q
Sun	Jacobi	1	1	2	1
	Red-Black	1	1	1	1
	RESID	2	1	3	1
SGI	Jacobi	2	1	2	1
	Red-Black	1	1	1	1
	RESID	3	1	3	1

提案するコスト関数$Cost_{New}$を用いる際に、PとQの値を3.2節に従って求める必要がある。対応するPおよびQの値を、表3に示す。

4.2 評価結果

表4、および表5に各マシンにおけるオリジナルのコードに対する性能向上率の平均値、およびL1/L2キャッシュミス回数削減率の平均値を示す。平均値は各問題サイズにおける値の算術平均である。また、L1/L2キャッシュミス回数削減率とは、オリジナルコードのミス回数からブロッキングを行ったコードのミス回数を引き、それをオリジナルコードのミス回数で割ったものである。

各表において、"C_Riv"はRiveraらのコスト関数$Cost_{Riv}$を用いて最適なブロックサイズを選んだ場合の性能を示している。一方、"C_New(L1)"は提案するコスト関数$Cost_{New}$を用いて最適なブロックサイズを選んだ場合の性能を示しており、$Cost_{New}$のLの値として、L1キャッシュラインサイズの要素数(ラインサイズ16Bなら$L = 2$)を代入したものである。また、"C_New(L2)"はLにL2キャッシュのラインサイズを採用している。

まず、キャッシュブロッキングの効果について議論する。表中、$Cost_{Riv}$を採用した"Euc3D"アルゴリズムを除いた全ての場合において、オリジナルのコードに比べブロッキングにより高性能が得られている。また、L1/L2キャッシュミス回数も、オリジナルに比べブロッキングにより大きく減少している。このことより、3次元PDE solverにおいて、キャッシュブロッキングが有効であることがわかる。

次に、$Cost_{Riv}$を用いた場合と、$Cost_{New}$を用いた場合の性能を比べる。表4の結果では、C_RivとC_New(L1)において、ほとんど性能差はない。これはSun Ultra30のL1キャッシュラインサイズが16Bと小さく、キャッシュトラフィック増加の影響がほとんどないためである。しかし、表5では、C_New(L1)はC_Rivに比べて全体的に良い性能が得られている。これは、SGI O2では、L1キャッシュラインサイズが32Bと大きくなるため、キャッシュラインがキャッシュトラフィックへ与える影響が大きくなるためである。ラインサイズが大きくなると、キャッシュラインサイズの効果を無視する$Cost_{Riv}$では誤差が大きくなるのに対し、$Cost_{New}$では$Cost_{Riv}$より正確にトラフィックを表すことがわかる。

表4　Sun Ultra Sparc 2での評価結果 (問題サイズ: 400-600)

Problem	Average Improvement	Euc3D			GcdPad			Pad		
		C_Riv	C_New(L1)	C_New(L2)	C_Riv	C_New(L1)	C_New(L2)	C_Riv	C_New(L1)	C_New(L2)
Jacobi	Perf	3.6%	3.6%	21.1%	24.9%	24.9%	29.0%	20.1%	20.2%	28.5%
	# L1 Miss	21.1%	21.1%	41.1%	59.2%	59.2%	57.2%	53.8%	56.2%	59.5%
	# L2 Miss	26.6%	26.6%	34.0%	34.3%	34.3%	34.0%	35.7%	36.0%	35.9%
Red-Black	Perf	37.0%	37.0%	59.6%	74.3%	74.3%	74.3%	62.2%	62.7%	74.5%
	# L1 Miss	41.1%	41.1%	55.6%	75.1%	75.1%	75.1%	60.9%	63.4%	62.6%
	# L2 Miss	70.8%	70.8%	71.0%	71.0%	71.0%	71.0%	72.7%	72.7%	72.7%
Resid	Perf	-3.8%	6.2%	11.1%	17.0%	17.0%	23.4%	12.0%	18.5%	30.0%
	# L1 Miss	42.4%	49.7%	49.5%	58.9%	58.9%	63.8%	51.4%	59.5%	60.9%
	# L2 Miss	10.9%	23.6%	24.6%	24.7%	24.7%	25.3%	26.6%	27.1%	26.8%

表5　SGI O2での評価結果 (問題サイズ: 200-400)

Problem	Average Improvement	Euc3D			GcdPad			Pad		
		C_Riv	C_New(L1)	C_New(L2)	C_Riv	C_New(L1)	C_New(L2)	C_Riv	C_New(L1)	C_New(L2)
Jacobi	Perf	37.7%	46.6%	48.8%	47.9%	50.8%	50.8%	48.9%	52.9%	54.0%
	# L1 Miss	17.0%	24.0%	24.4%	30.8%	35.6%	35.6%	22.0%	26.6%	28.2%
	# L2 Miss	29.1%	34.3%	34.8%	36.8%	36.5%	36.5%	36.1%	36.7%	36.8%
Red-Black	Perf	174%	175%	180%	158%	158%	161%	184%	184%	184%
	# L1 Miss	57.8%	57.8%	58.6%	68.1%	68.1%	70.4%	48.8%	55.2%	55.3%
	# L2 Miss	70.2%	70.2%	70.3%	72.0%	72.0%	71.6%	71.1%	71.1%	70.4%
Resid	Perf	10.8%	31.0%	32.2%	30.8%	36.2%	36.4%	38.0%	45.1%	45.7%
	# L1 Miss	24.1%	29.9%	29.6%	34.2%	36.6%	35.3%	28.4%	32.5%	33.9%
	# L2 Miss	22.2%	28.3%	29.1%	28.0%	29.5%	29.3%	26.7%	29.3%	29.7%

さらに、C_RivとC_New(L2)を比較すると、その性能差は顕著になる。C_New(L2)を用いた場合、C_Rivに比べSunでは平均11.5%の高性能を達成しており、SGIでは7.0%の高性能を達成している。L2キャッシュはL1キャッシュに比べて、大きなラインサイズを採用していることから、キャッシュラインがキャッシュトラフィックへ与える影響が大きく、我々のコスト関数を用いることで、キャッシュトラフィックを最小限に抑えることができたためと考えられる。

今回の評価では、64BのL2キャッシュラインを持つマシン上で評価を行った。現在、64B以上のキャッシュラインを採用しているマシンも少くない。例えば、SGI Origin2000では128BのL2キャッシュラインを持っている。今後の半導体の技術動向を考えると、L2キャッシュのラインサイズは、長いメモリアクセスレーテンシ隠蔽のためより大きくなることが予想される。そのような大きなラインを持つマシンでは、今回の評価結果よりもキャッシュトラフィック増加の影響は大きくなるため、ラインサイズの影響を考慮しつつ、性能最適化を行うことが今後より重要になると考えられる。

5. まとめと今後の課題

HPCにおいて3次元のPDEを解くことの必要性は、今後も増加していくと考えられる。3次元の問題では、メモリアクセスがボトルネックとなることによる性能低下が顕著でるため、主記憶とキャッシュ間のデータ転送をできるかぎり削減することが、高性能を得るためには特に重要である。

このことから、本稿では3次元PDE solverにおける最適なブロックサイズを選択するための新たなコスト関数を提案した。提案するコスト関数はキャッシュラインの影響を考慮しており、従来の手法に比べキャッシュトラフィックを小さく抑えることが可能である。

性能評価結果では、提案するコスト関数を用いることで、従来のコスト関数を用いた場合に比べ、キャッシュミス回数を削減できること、またそれにより高性能を得られることがわかった。このことから、我々の提案するコスト関数は、従来のものより優れていると結論付けることができる。

今後の課題としては、ラインサイズの異なる他のマシンで性能評価を行うことや、実アプリケーションに対して、本コスト関数を用いたブロッキングを適応し、評価を行うことなどが挙げられる。

謝辞　本研究を行なうにあたり、御助言、御討論頂いたTEAグループの皆様、ならびに東京大学南谷崇教授に感謝致します。なお、本研究の一部は日本学術振興会未来開拓学術研究推進事業「計算科学」(Project No. JSPS-RFTF 97P01102)、および、文部科学省科研費特定領域研究(A)「知的瞬時処理複合化集積システム」によるものである。

参考文献

1) M. Lam, E. Rothberg and M. Wolf, "The cache performance and optimizations of Blocked Algorithms", Proc. ASPLOS-IV, pp.63-74, 1991
2) http://www.specbench.org/
3) D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0", NASA Ames Research Center Report, NAS-05-020, 1995.
4) G. Rivera and C.-W. Tseng, "Tiling Optimizations for 3D Scientific Computations", In Proceedings of Supercomputing 2000, November, 2000.
5) G. Rivera and C.-W. Tseng, "A Comparison of Compiler Tiling Algorithms", In Proceedings of the 8th International Conference on Compiler Construction (CC'99), March 1999.
6) S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout", Proc. of PLDI, June 1995.