一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
CPSY2014-52(2014-10)

ブロックデバイス非同期 I/O による フラッシュストレージを用いたステンシル計算の性能評価

丹 英之†§　　緑川 博子†§

†成蹊大学理工学部 〒180-8633 東京都武蔵野市吉祥寺北町 3-3-1
§ JST CREST
E-mail:　†hidetan@ci.seikei.ac.jp, midori@st.seikei.ac.jp

あらまし　物理メモリを超えた問題サイズのステンシル計算を実行するため, テンポラルブロッキングによりデータ参照局所性を高めるアルゴリズムと共に，フラッシュストレージを拡張メモリとして用いる 3 種類の手法について検討した．入出力状況から，フラッシュストレージの特性に合う非同期 i/o 方式(aio)が有利であるが，デバイスのブロックサイズに依存したバウンダリ制約が生じ，演算時間に影響を及ぼすことがわかった．そこで，aio 方式の最適化を検討ところ，block で用いる 3 次元配列のメモリ領域は連続して確保すること，そして，x-y 平面の領域間にページパディングを入れること，更に，内部ループブロックの形状を変更することで，既存の実装に比べ実行時間を約 45%削減することが出来た．

キーワード　不揮発性メモリ，フラッシュストレージ，ステンシル計算，テンポラルブロッキング，非同期 I/O

An Evaluation of Stencil Computation using Flash Storage with Asynchronous Block Device I/O

Hideyuki TAN†§　　Hiroko MIDORIKAWA†§

†Seikei University　3-3-1 Kichijoji-Kitamachi, Musashino-shi, Tokyo, 180-8633 Japan
§ JST CREST
E-mail:　†hidetan@ci.seikei.ac.jp, midori@st.seikei.ac.jp

Abstract　In order to execute stencil calcurations of the large scale problems beyond physical memory, we evaluated three different implementations with the temporal blocking locality-aware algorithm that uses a flash storage as an expanded memory. We also proposed an asynchronous i/o (aio) technique tuned for flash storages by analyzing a situation of i/o. However, compared with other techniques, the execution time of the aio technique is long because of boundary restrictions. Then, we investigated the optimization of the aio technique, and the reduced execution time by 45% by allocating blocks in a continuation memory space, padding page between x-y plain memory areas, and inner loop transformation.

Keywords　non-volatile memory, flash storage, stencil computation, temporal blocking, asynchronous i/o

1. はじめに

昨今の科学技術分野における数値シミュレーションは，非常に重要な位置づけとなっている．その数値シミュレーションの中では，近傍格子との関係を計算するステンシル系の計算カーネルがよく利用されている．数値シミュレーションを実世界と結びつけ，社会の基盤・進歩の礎とするには，より大きな規模，より高い解像度でのシミュレーションが要求される．ステンシル計算にて大規模，高解像度の問題を解くには，前の時間ステップにおける近傍格子の値を要することから，問題が収まるサイズの記憶装置が必要になる．

主記憶装置には，一般的に DRAM が用いられているが，大抵の場合，計算機に搭載できるメモリサイズには制限があること，また，DRAM の仕組み上，記憶保持のリフレッシュに電力を要することから，多数ノードからなるクラスタ型のスーパーコンピュータなどで，各ノードに大量の DRAM 搭載することは消費電力の観点からも難しい．

そこで我々は，搭載している物理メモリサイズよりも大きな問題を解く際に，不揮発性メモリである NAND-Flash ストレージを，「アクセス性能は劣るが大容量なメモリ」として用いる手法を探ってきた[1][2][3]．また，ステンシル計算を対象に，データ参照局所性を高めるアルゴリズムを導入し，DRAM と Flash ストレージによるメモリ階層を効率よく利用して，物理メモリを越えるサイズの問題を高速に処理す

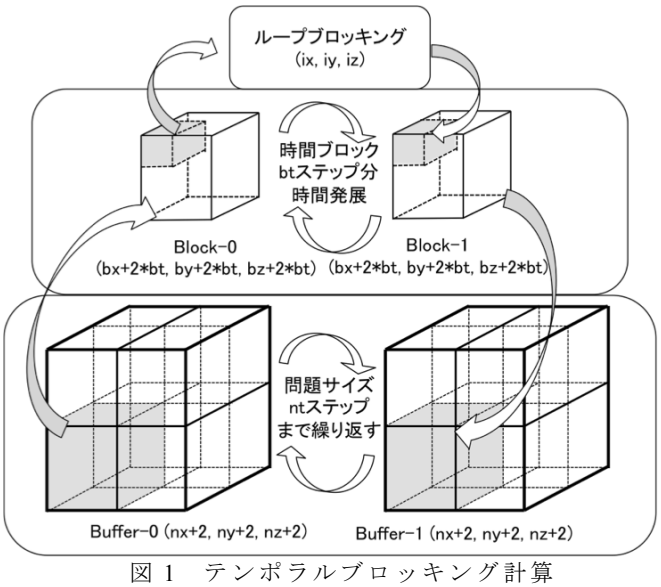

図 1　テンポラルブロッキング計算

る手法を提案した[4][5].

これまで, Flash ストレージを拡張メモリとして用いるために，1)スワップデバイスとして利用する swap 方式，2)ファイルシステムとして利用し，ファイルマップする mmap 方式, 3) ブロックデバイスを直接参照し，非同期入出力を行う aio 方式，の 3 つの方法を試みた．この中で，aio 方式は，プログラム内で明示的な i/o を用いており, DRAM と Flash ストレージ間のデータ転送を大きなサイズで一度にまとめて行うことができ，性能的に有利である．しかし，aio では，i/o バッファの開始アドレス，i/o サイズ，デバイス上のオフセットがブロックサイズ境界となっている必要があり，プログラム上，様々な制約が生じる．

本稿では，このような aio 方式の制約下で，アプリケーション性能を向上させるための手法を検討する．

2. テンポラルブロッキングと Flash ストレージ

2.1. テンポラルブロッキングとメモリ階層

ステンシル計算では，対象となる格子点の近傍格子を参照しつつ演算し，その格子点を更新する．これを時間ステップ毎に問題サイズ分の格子を全て走査することで系の時間発展が進行する．

一般には，空間ブロッキングを行うことで，部分領域にある格子の更新をまとめて行う操作を，繰り返す処理として実装される．これによりデータ参照の局所性を作り出すことで性能向上を図られる．しかし，時間ステップでの繰り返しに着目すると，更にデータ参照の時間的局所性を向上させることができる．これがテンポラルブロッキング(時間ブロッキング)である[6].

テンポラルブロッキングでのデータと処理の流れを図 1 に示す. Buffer-0 が問題サイズ(nx+2, ny+2, nz+2)格子の初期値が格納されているメモリ領域であり，格子点の更新結果格納先として Buffer-1 がある．問題サイズの格子は各軸方向の前後で 1 要素ずつ境界領域をとっているので, 各々指定サイズ+2 となる. ここから，部分領域の格子データを Block-0 に取り込み，その領域だけでステンシル計算を行うが，同時に時間ブロック bt ステップ分の計算も行う．Block-0 に取り込む格子データは，時間ブロックを考慮しない場合，空間ブロック(bx, by, bz)格子分だけでよい．しかし，その部分領域だけで bt ステップの時間を発展させるには，その bt ステップ分の演算で参照する近傍格子が必要になる．近傍格子は各軸方向前後に bt 要素必要である．このため, Buffer から Block へ取り込む格子データは，(bx+bt*2, by+bt*2, bz+bt*2)格子分となる．ステンシル計算での時間発展で Block-0, Block-1 のメモリ領域を，それぞれ格子データの入力元，出力先として交互に用い，bt ステップの時間発展が完了したら，その結果を Buffer-1 に戻す．これを Buffer-0, Buffer-1 でも同様にシミュレーションの時間ステップ nt に達するまで繰り返す．Block では，bt ステップ分本来の格子よりも大きい格子を更新処理するという冗長計算を含む．

メモリはデータアクセスの速度によって階層化される．今回の構成では，CPU キャッシュ，メインメモリの DRAM,そして NAND の Flash ストレージの 3 階層である．そこで，テンポラルブロッキングで用いる Block のメモリ領域を DRAM に，Buffer のメモリ領域を Flash ストレージに配置するようにした. また, Block でのステンシル計算では，CPU キャッシュを考慮し，格子走査の 3 重ループに対し，ループブロッキングを行った．

2.2. メインメモリ-Flash ストレージ間の i/o 方式

メインメモリと Flash ストレージ間でのデータ転送方式は 3 通りある．

(1) swap 方式 : Flash ストレージをスワップデバイスとして指定する方法である．プロセス内でメモリを確保すると，OS の仮想メモリとして Flash ストレージが使われる．しかし，アプリケーションプログラムからは，ページイン・ページアウトの制御は出来ない．そこで，Block として確保した領域を mlock(2)でページアウト禁止としてメモリ階層を作った．

(2) mmap 方式 : Flash ストレージにファイルシステム ext4fs を載せ，Buffer-0, Buffer-1 に対応するファイルを配置し，mmap(2)にてプロセスのメモリ空間としてマップする方法である．

(3) aio 方式 : Buffer-Block 間を Linux ネイティブの libaio を用い，アプリケーションプログラムから明示的に非同期で入出力する方法である. Flash ストレージをブロックデバイスとして直接入出力を行う．

ノンブロッキングの i/o なので, Block のステンシル計算を行いながら i/o を行うようにしている．i/o はブ

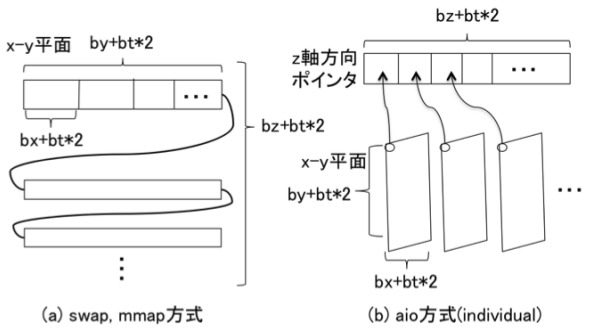

図 2　メモリ上の空間ブロックのレイアウト

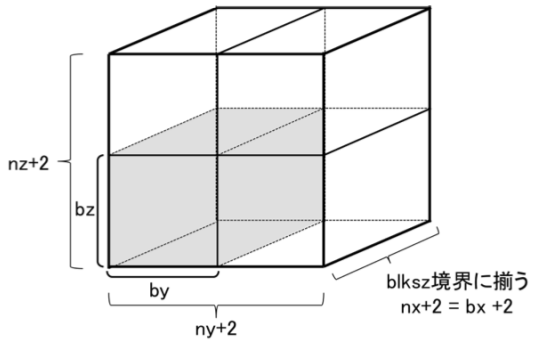

図 3　問題サイズと空間ブロックサイズの関

ロックの前半・後半の 2 回に分け，入力時はブロック全体の read を発行後，ブロック前半分の read が完了したことを確認してから前半分の計算を開始し，ブロック後半分の read 完了を待ってから後半分の計算を開始するようにした．また，出力時は，ブロック前半分の計算終了後，前半分の write を発行してから，後半の計算を開始するようにした．後半の計算終了後，後半分の write を発行し，ブロック全体の write が完了したことを確認してから次の時間・空間ブロックの処理に移る．

libaio で非同期 i/o を実現するには，ブロックデバイスを open(O_DIRECT)で開くことが必要である．このため，i/o バッファの開始アドレス，i/o サイズ，デバイス上のオフセットを，ブロックデバイスのブロックサイズ(以後,blksz と呼ぶ)境界に合わせる必要がある．これらの制約により，aio 方式では他の 2 つの方式に比べ，実装がやや複雑になる．

2.3. 空間ブロックのデータ構造

メモリ上の空間ブロックのレイアウトを図 2 に示す．swap, mmap 方式では，Block のメモリ領域は空間・時間ブロック分の 3 次元格子が入るサイズの領域を一括で確保した．x-y 平面(bx+bt*2, by+bt*2)の領域が連続して z 軸方向に bz+bt*2 枚並んでいる．

aio 方式でも，ブロックのパラメータから問題サイズを決めることで, i/o サイズとデバイス上のオフセットが blksz 境界に一致するメモリの一括確保ができる．しかし，この問題サイズの決め方は汎用性が低い．そこで，問題サイズに下記の制約だけを課し，自由度を向上させつつ，blksz 制約を満たすことにした．

(1) 問題の x 軸方向サイズ nx と空間ブロックの x 軸方向サイズ bx が等しいこと．

$$nx = bx$$

問題サイズ格子外は，値が無いものとして計算を省略している．このため，x 軸方向の空間ブロッキングをやめると，x 軸の空間ブロックに時間ブロックの bt ステップ分を追加する必要がなくなる．

表 1 実験環境

部位	概要
CPU	Xeon E5-2650 2.00GHz x1 (8cores)
Memory	DDR3-1600 ECC 8GiB x4 (32GiB)
Flash Storage	ioDrive2 MLC(785GB)
Distribution	CentOS 6.4 (x86_64)
Kernel	3.13.0
Compiler / option	gcc version 4.4.7 20120313 / -O3

(2) 境界領域も含めた問題の x 軸方向サイズは，blksz の倍数となるメモリを必要とする要素数であること．

$$((nx + 2) \times sizeof(element\_data\_type))\%\ blksize = 0$$

(nx+2)は，図 1 に示すように，問題サイズの格子は各軸方向の前後で 1 要素ずつ境界領域を加える必要があることから nx に 2 を加えている．

制約は，問題の x 軸方向サイズ nx と空間ブロックの x 軸方向サイズ bx は一致していること，且つ，blksz の倍数になるメモリサイズに一致する要素数であることの 2 つである．これらの制約により，aio のバウンダリ制約を満たすことができる．問題サイズと空間ブロックサイズの関係を図 3 に示す．

例えば，blksz が 4KB，要素の型が double(8Bytes)とすると，問題サイズと空間ブロックの x 軸要素数は 512 の倍数であることが制約になる．問題サイズ格子の各軸方向の前後で 1 要素ずつ領域も考慮する必要があるので，実際には nx+2=bx+2 で，且つ 512 の倍数となっている必要がある．本稿で計測に用いた問題及び空間ブロックの x 軸方向サイズは 2046 とした．

3. i/o 方式と処理時間

上記の制約下，同じ空間ブロックサイズを用いて，ステンシル計算を，swap 方式，mmap 方式，aio 方式の 3 方式で実行した際の，io 状況，CPU 利用率などについて解析した[5].

近傍 7 点ステンシル計算について，格子サイズ 2046x2048x1024(問題サイズ 64GB), 256 タイムステップの問題を，空間ブロック 2046x512x512 格子，時間ブ

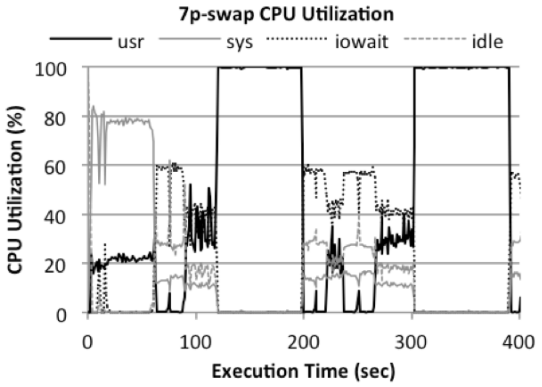

図 4　swap 方式での CPU 利用率

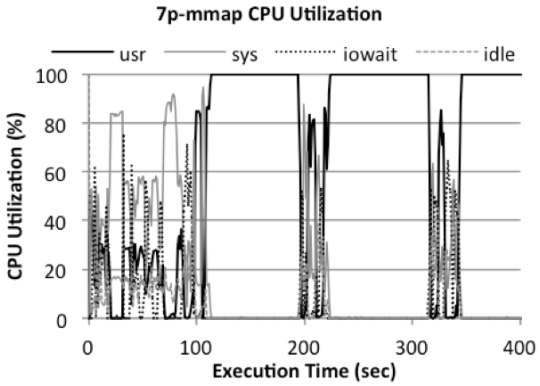

図 5　mmap 方式での CPU 利用率

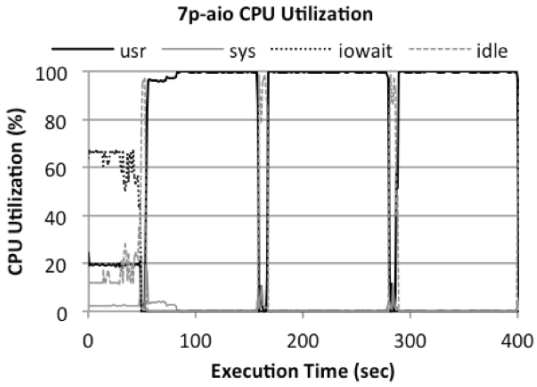

図 6　aio 方式での CPU 利用率

ロック 128 ステップ，内部ループブロック 32x32x32 格子のパラメータで処理している．実験環境を表 1 に示す．Flash ストレージには PCIe 接続の SSD である ioDrive2 を用い，ブロックサイズ 4KB で物理フォーマットして用いた．また，mmap 方式でのファイルシステムには ext4 を用いた．aio 方式では問題サイズを調整し，図 2(b)ではなく，図 2(a)の，swap, mmap 方式と同様，Block のメモリ領域を一括で確保した．

図 4, 5, 6 は，実行時間 400 秒までの iostat で得られたユーザ(usr)，システム(sys)，io 待ち(iowait)，アイドル(idle)の CPU 利用率を示す[5].

aio 方式では，Buffer-Block 間のデータ入れ替えが短

表 2 各方式での CPU 利用率詳細

方式	aio	mmap	swap
実行時間 (秒)	1847.55	1929.86	2720.58
実効性能 (MFlops)	5945.38	5691.79	4037.51
CPU Utilization 1 周期平均時間(秒)	114.67	118.47	169.13
1 周期中の user <90%	7.73	29.73	85.27
平均時間(秒)	6.7%	25.1%	50.0%

時間で終了していることがわかる．一方，swap 方式，mmap 方式では，計算しながら入れ替えが発生しているため，aio 方式に比べ長時間に渡って小さいサイズの i/o が発生していることを確認できる．このため，CPU を 100%利用できている期間が短くなっている．特に swap 方式では，I/O 待ちが目立ち，効率良く CPU を利用できていないことが伺える．mmap 方式では，swap 方式ほど長い I/O 待ちではないが，aio 方式に比べると，効率が悪い．

プロセスのユーザ CPU 利用率が 100%となる部分がブロックの演算を処理している期間である．空間ブロッキングで 8 分割，時間ブロッキングで 2 分割，合計 16 回のブロック演算が実行されるので，1 回のシミュレーション実行で図中 usr の示す台形は, 16 個現れる．各方式での台形の周期，及び io 待ちの時間を表 2 にまとめた．表中の実行時間とは，問題データの初期化時間を除いたステンシル計算の演算時間である．CPU 利用率が連続して 90%以上である期間をブロック演算時間とし，あるブロック演算の終了から次のブロック演算の終了までを 1 周期とした．また，1 周期中 CPU 利用率が 90%未満であった期間を io 待ち時間に相当するとした．aio 方式と mmap 方式では CPU 利用率が 90%に満たない期間の差が 22 秒であるにもかかわらず, 平均周期の差が 4 秒程となっている．

つまり, aio 方式では mmap 方式に比べ演算時間が長くなっている．この原因は空間ブロックの境界アライメントなど aio 方式での制約に由来することが分析の結果わかってきた．

4. aio 方式での性能向上

4.1. 空間ブロックデータ構造の最適化

aio 方式において Block のメモリ領域を確保する際，図 7 に示す 3 方式を用いて比較することにした．

第 1 の方式は，図 2(b)のように，x-y 平面分のサイズを 1 枚ごとに blksz でアライメントして確保する方式である．posix_memalign(3)で blksz にアライメントした x-y 平面(bx+bt*2, by+bt*2)のメモリ領域の開始アドレスを z 軸方向のポインタ配列に格納している．各

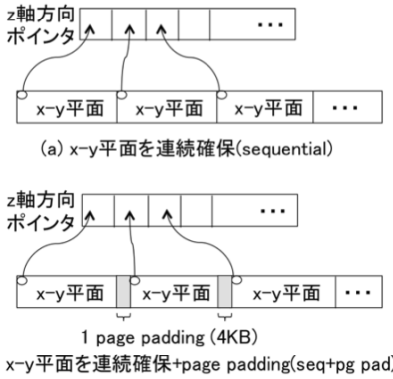

図 7　aio 方式でのメモリ配置とページパディング

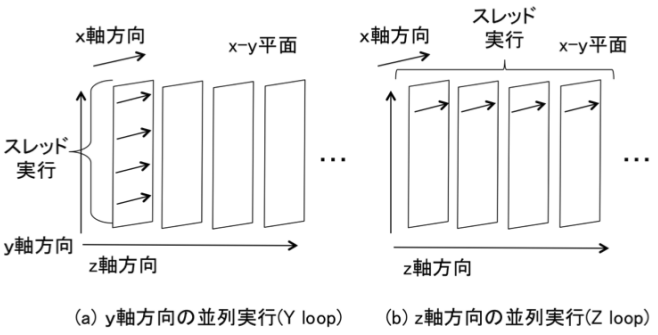

図 9　ループブロッキングのワークシェア

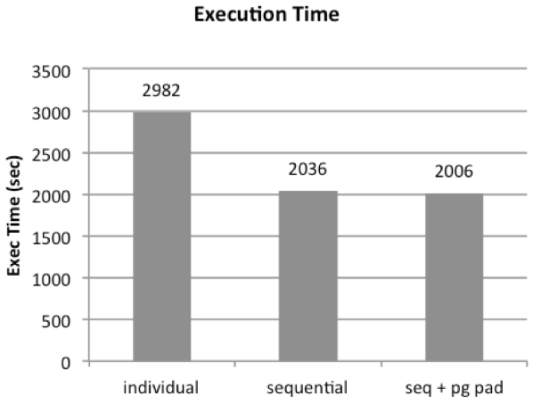

図 8　メモリ配置変更での実行時間

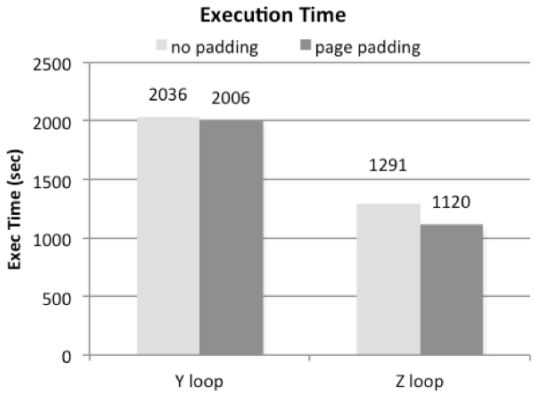

図 10　ループブロッキング形状と実行時間

x-y 平面の開始アドレスはバラバラになり，不連続な領域が複数確保される．

第 2 の方式は，図 7(a)のように，全ての x-y 平面分を一括して確保し，blksz 境界に一致するアドレスを各々の x-y 平面の開始アドレスとなるように z 軸方向の配列に格納するようにした．これは，2.2 節で述べた x 軸方向のサイズに制約を設けることで可能になる．

第 3 の方式は，図 7(b)（seq+pg pad)のように，独立して i/o を行う領域間にページパディング(ページサイズ 4KB)を入れる方式である．これは，大きなアドレス空間を複数スレッドで参照する場合，TLB のエントリコンフリクトを軽減する効果があると考えられ，ページパディングは 2.2 節で述べた制約にも反しない．

マルチスレッド計算では，多次元配列の低次元要素にパディングすることで，複数スレッドによる同一メモリバンクへの同時アクセスを回避する手法がよく用いられる．mmap、swap の 2 手法ではこれを導入することが可能で，x 次元サイズを 1 増やすことで,マルチスレッド計算部分の性能向上に効果があることがわかっている．しかし aio 方法では，blksz 境界制約や，i/o をおこなう Buffer 格子データも空間ブロックに対応して連続している必要があるなどの制限のため，バンクコンフリクト回避を目的とした 1 要素のパディングは難しい．

3 節と同様の近傍 7 点ステンシル問題(問題サイズ 64GB)を，個々の x-y 平面の領域を個別に確保する既存方式(図 2(b), individual)，複数の x-y 平面の領域を連続確保した方式(図 7(a), sequential)，更に，連続確保した上に x-y 平面間に 1 ページパディングした方式(図 7(b), seq+pg pad)を実行した．内部ループブロック形状は，x 方向を長くして，2048x64x1 格子としている．3 種のデータ配置による実行結果を図 8 に示す．

sequential 方式は，individual 方式の約 70%の実行時間に短縮された．ページパディングの効果は，メモリ確保方法の違いほど大きくないが，若干の改善が見られた．空間ブロックを連続領域にすることで，プロセスの仮想メモリ空間が単純になりメモリ管理コンストが低下すること，もしくは，プリフェッチなどのハードウェア機構が有利に働くからと考えられる．

4.2. 内部ループブロッキングの形状とワークシェア

図 1 に示す通り，Block 間でのステンシル計算は，CPU キャッシュと DRAM のメモリ階層を考慮し，ループブロッキングを行うことで，データ参照の局所性を向上させている．このループブロッキングで，OpenMP によるワークシェアを行っている．これまでの調査から，内部ブロックサイズは L3 キャッシュサイズに対応させ，形状は連続アドレスを生かすように x 方向を長くするほうが性能がよいことがわかってい

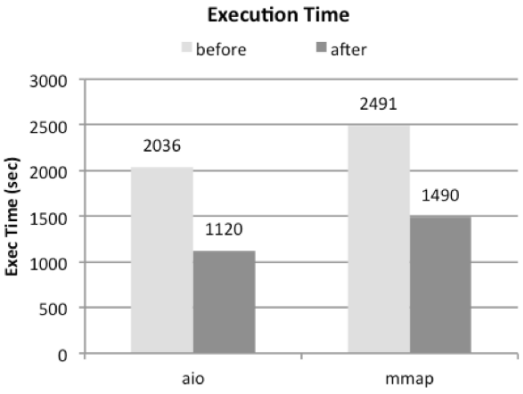

図 11　aio, mmap 方式での最適化前後の実行時

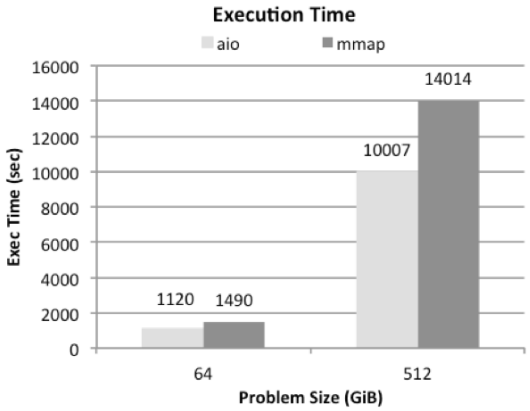

図 12　問題サイズ 512GB での実行時間

る．スレッド間のワークシェアは，当初，図 9(a)のように y 軸方向で並列に行っていた(Y loop)．これを，図 9(b)のように z 軸方向で並列に行うようにした(Z loop)．Y loop では，x-y 平面 1 枚を複数スッドで処理していく．一方，Z loop では，各スレッドが各々の x-y 平面を処理していく．前節 sequential 方式を用いて，この２つの方式の性能を比較した．

Y loop, Z loop をページパディング有無(+pg pad)の組合せで実行した際の，実行時間を図 10 に示す．問題サイズは前節と同じで，Z loop では，内部ループブロックの形状を 2048x1x64 格子としている．Z loop は，Y loop に比べ，約 60%の実行時間となった．またページパディングの効果が，Y loop では 2%程度の差だったものが，Z loop では，約 13%実行時間が短くなった．

5. 最適化後の性能評価

これら調査により，aio 方式と mmap 方式の最善手法として以下を用い，性能を比較した．

aio 方式:x-y 平面間に 1 ページパディングし連続確保，内部ループブロックを z 軸方向でワークシェア．

mmap 方式:x 軸方向に 1 要素パディング，内部ループブロックを z 軸方向でワークシェア．

これを，問題サイズ 64GB のパラメータで実行した際の実行時間を図 11 に示す．最適化を施す前と比較すると，aio 方式(図 7(a)sequential)では 55%，mmap 方式(要素パディング無し，y 軸方向ワークシェア)では 60%，まで実行時間が短縮できた．また，最適化により，aio 方式は mmap 方式の 75%の実行時間となった．

更に，物理メモリ 32GB の 16 倍の問題サイズでの評価も行った．格子サイズは 2046x4096x4096(問題サイズ 512GB)で，この問題を aio 方式，mmap 方式による実行時間を図 12 に示す．

6. まとめ

境界制約のある aio 方式でのテンポラルブロッキングにおける実行時間短縮のための手法を検討した．

問題と空間ブロックの x 軸方向サイズを blksz に揃えるが，このとき by,bz,bt は DRAM 物理メモリにはいるだけ大きくするほうが有利である．そして，空間ブロックは連続領域で確保し，ページパディングをすることが効果的であった．また，内部ループブロックはキャッシュのサイズに合わせた形状であること，そして，x 軸方向を大きくし，各スレッドのタスクは小さすぎない大きさであること，そして，コアがキャッシュを汚し合わないようなワークシェアが有利であることが確認できた．

今後は，3 次元配列を aio 方式で容易に扱えるようなミドルウェアを検討していきたい．

文　献

[1] 緑川博子,丹英之:"メモリサイズを越えるデータ処理を目的としたバス接続型 SSD の性能評価",情報処理学会，ハイパフォーマンス研究会,Vol.2013-HPC-140, No.44, pp.1-6, (2013.8)

[2] 丹英之,緑川博子:"フラッシュ向け Linux スワップシステムの評価",電子情報通信学会,コンピュータ・システム研究会 Vol.113, No.282, pp.61-66, (2013,11)

[3] 丹英之,緑川博子："フラッシュ SSD をメモリセマンティクス API で用いるための予備調査",ハイパフォーマンスコンピューティングと計算科学シンポジウム HPCS2014, HPCS2014 論文集,(2014,1)

[4] Hiroko Midorikawa, Tan,Hideyuki, Toshio Endo, ” An Evaluation of the Potential of Flash SSD as Large and Slow Memory for Stencil Computations”, IEEE The 12th International Conference on High Performance Computing & Simulation, HPCS2014, 2014, pp.268-277,

[5] 緑川博子，丹英之: "大規模ステンシル計算のための Flash SSD 向けテンポラルブロッキングの性能評価",情報処理学会,ハイパフォーマンスコンピューティング研究会,Vol.2014-HPC-145, No.22, pp.1-9, (2014.7)

[6] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann and Holger Fehske, “Efficient temporal blocking for stencil computations by multicore - aware wavefront parallelization”, Computer Software and Applications Conference, vol.1, pp. 579‐586, 2009.