分散共有メモリを用いた並列FFTとその最適化

額田 彰[†] 西田 晃[†] 小柳 義夫[†]

本研究では，高速なマイクロプロセッサItaniumを搭載した分散共有メモリシステムNEC Itanium ccNUMAサーバ（AzusA）上で並列FFT（Fast Fourier Transform）アルゴリズムを実装し，2^{24}点FFTの計算において8 PEで3.12 Gflops（ピーク性能の13.3%）という高い性能を引き出すことができた．分散共有メモリアーキテクチャで重要となるデータの配置方法の違いによる性能差を分析し，適した配置方法を選択した．また従来のキャッシュメモリを有効利用するFFTアルゴリズムに改良を加えin-placeアルゴリズムに対応させた．これにより使用するキャッシュメモリ量が少なくなり，より大きなサイズのFFTを計算する場合においても高い性能を出すことができる．

Parallel Implementation of FFT Algorithms on Distributed Shared Memory Architecture and Its Optimization

Akira Nukada,[†] Akira Nishida[†] and Yoshio Oyanagi[†]

In this study, we implemented parallel FFT (Fast Fourier Transform) algorithm on a distributed shared memory system, NEC Itanium cc-NUMA server (AzusA). We achieved 2.88 Gflops with 8 processors (12.4% of peak) for computing 2^{24}-point FFT. On distributed shared memory systems, data placement is important for high performance. Therefore, we have to use proper data placement. And we improved the conventional algorithm that is suitable for shared memory systems. In our algorithm, we can use in-place FFT algorithms, and can compute FFT of larger size on limited cache memory.

1. はじめに

近年，並列アーキテクチャ技術の発展により，高性能なマイクロプロセッサを複数搭載した共有メモリ型並列計算機が比較的容易に利用できるようになってきた．共有メモリ型アーキテクチャは，プロセッサの接続形態によってバス結合型とネットワーク結合型に大別することができる．バス結合型アーキテクチャでは，バスの帯域幅による制約から，構築可能な並列環境の規模が限られているため，帯域幅の限界に近づくと並列化効率がプロセッサ台数に比例しにくくなる．これに対して，主記憶を分散配置するネットワーク結合型の分散共有メモリアーキテクチャでは，主記憶に対するアクセス時間はそのプロセッサと同じノード内であるか他のノードにあるかで不均等になる．しかし一般に大規模な計算では各プロセッサのメモリアクセスに局所性がある場合が多く，このような場合には各ノードの主記憶へのデータ配置方式を工夫することによってより多くのメモリアクセスがノード内で済むようになる．ノード内で済むメモリアクセスの割合が多い場合にはプロセッサ数が増えても高いスケーラビリティを持つ可能性が高くなり，高い拡張性が必要となる大規模科学技術計算に適した形態であると考えられる[1)]．

コモディティハードウェアによって構築された分散共有メモリ型アーキテクチャを採用することにより，容易に資源の拡張が可能な計算機環境を実現することを目標とするとともに，共有メモリアーキテクチャにおいてユーザがアーキテクチャの下位構造を意識することでより高性能な科学技術演算を行うことができる．

本稿では，NEC Itanium ccNUMAサーバ上に実装されたLinuxオペレーティングシステムを利用し，並列FFTアルゴリズムのデータ配置方法とその最適化手法について，実機上で様々なデータを元に検討する．また従来のキャッシュを有効利用するアルゴリズムにはキャッシュ内で行われているFFTアルゴリズムにStockham FFT等のbit-reverse処理が必要ないアルゴリズムが用いられることが多いが，このbit-reverse処理を主記憶アクセス時間に隠蔽することでin-place

† 東京大学大学院情報理工学系研究科コンピュータ科学専攻
Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo

アルゴリズムを用いることができるような手法を提案し，実際にほぼ隠蔽可能であることが確認できた．

2. 評価環境

2.1 IA-64 Intel Itanium Processor

現在，Intel 社および Hewlett Packard 社により開発が進められている IA-64 アーキテクチャは，Intel の 32 bit アーキテクチャ（IA-32）との互換性を維持してはいるものの，多くの新しい機能を搭載した 64 bit アーキテクチャである．これにより，プロセッサの構造は複雑となるが，標準的な RISC プロセッサと比べてより汎用性が高く，低価格での供給を可能にしている．また，適切なチップセットを用いることにより，最大 512 CPU までメモリを共有することができる．IA-64 命令をサポートした最初のプロセッサが Itanium [2)]である．

Itanium は既存のプロセッサに比べて多くの複雑な機能を持っている．まず VLIW 型の命令形式を採用しており，コンパイラ等で明示的に並列実行可能な命令群を束ねることで 1 サイクルあたりに処理する命令の数（IPC）を向上させることが可能となっている．

高性能科学技術計算に重要な機能としては，128 個という余裕があるレジスタ数，2 つの積和演算ユニット，ソフトウェアパイプライン技術による高い浮動小数点数演算のスループットがあげられる．また，プレディケーションレジスタやレジスタローテーションを用いることで，複雑なループ文もプロローグ部やエピローグ部を含めてコンパクトなコードサイズに収めることができる．しかしながら C や Fortran 等の高級言語で記述したやや複雑なコードでは依存関係がないことを示すことができない場合に，コンパイラだけでは十分に最適化が行われないこともありうる．特に各命令に要するレイテンシが他のアーキテクチャに比べて大きいため，この点は性能に大きく影響する．

Itanium のキャッシュメモリをまとめると**表 1** のようになる．浮動小数点数データは L1 Data キャッシュを経由しない，L1 キャッシュと L2 キャッシュにあるデータに包含関係がない等の特徴がある．キャッシュラインのリプレイス方式は hint により指定可能であるが，その中で一番実行が速くなる時間的局所性を用いる方式を用いた．

表 1 Itanium プロセッサのキャッシュメモリ
Table 1 Cachememory of Itanium Processor.

Level	Type	Latency
L1D	4-way, WT, 32 B line	Int:2
L1I	4-way, 32 B line	N/A
L2	6-way, WB, 64 B line	Int:6, FP:9
L3	4-way, WB, 64 B line	Int:21, FP:24

IA-64 アーキテクチャを用いた商用の分散共有メモリ型並列計算機の 1 つとして，NEC の AzusA Itanium サーバをあげることができる．本研究では，現在入手可能な IA-64 アーキテクチャベースの分散共有メモリ環境として AzusA を導入し，コモディティ分散共有メモリ上における計算性能について予備評価を行った．

2.2 NEC AzusA

ここで，評価に用いた計算機システム NEC AzusA について述べる．AzusA では，最大 4 個のプロセッサおよび主記憶を搭載するセルカード内でバスを共有し，セル間をクロスバスイッチで接続することによりコヒーレンシを保ち，最大で 16 個の IA-64 プロセッサを単一インスタンスのオペレーティングシステムで管理することができる．**図 1**，**図 2** に AzusA のシステムおよびチップセットの構成を示す[3)]．セル内の CPU–チップセット，メモリ–チップセット間の帯域幅はそれぞれ 2.1 GB/s，4.2 GB/s であり，セル–クロスバスイッチ間，クロスバの総合帯域幅はそれぞれ 4.2 GB/s，16.8 GB/s である．

Tag メモリ

AzusA では異なるノードにあるプロセッサ間でキャッシュの一貫性を効率良く保持するため Tag メモリ（図 2 では Tag SRAM と表記）という機構を用いている．

プロセッサから送出されるメモリアクセスリクエストは，まずバス内の他のプロセッサのキャッシュにヒットするかどうか検索され，またバスに接続された AzusA チップセットおよびアドレスネットワークを介して，リモートセル上のプロセッサのキャッシュに対してスヌープと呼ばれる検索を行う．ローカルセル内のプロセッサから送出されたメモリリクエストにヒットしたデータは，そのラインアドレスが Tag メモリと呼ばれるスヌープキャッシュメモリに記録され，リ

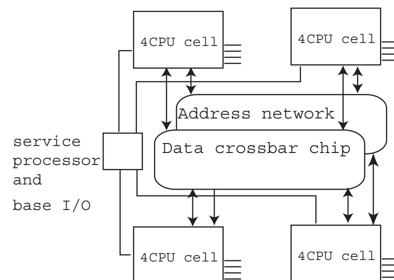

図 1 AzusA のブロック図
Fig. 1 AzusA system block diagram.

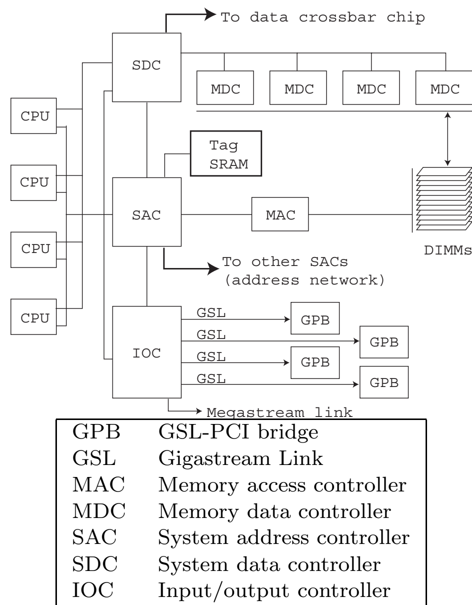

図 2 AzusA チップセットの構成
Fig. 2 AzusA chip set components.

モートセルからのスヌープリクエストが Tag メモリにヒットしたセルのみがそのリクエストを受け取る．データがローカルセルのメモリにマップされている場合には，スヌープ結果を待たずにメモリアクセスを開始し，スヌープ結果に応じて適切なデータをプロセッサバスに返す．以上の機構により，ローカルセルからのメモリデータ読み出しに対して 200 ns 以下，リモートセルからの読み出しに対して 300 ns 以下のメモリレイテンシを実現している．

2.3 評価に用いた環境

実際評価に用いた計算機は，NEC AzusA（733 MHz Intel Itanium Processor × 8, 32 KB L1 cache（16 KB data/16 KB instruction），96 KB L2 cache, 2 MB L3 cache, AzusA chipset, 2 GB main memory）．CPU あたりの CPU バスの帯域を重視しており，各セルに 2 CPU ずつ搭載した 4 セルの構成である．OS は Red Hat 7.1 ベースの NEC Linux R1.2, カーネルは Linux kernel 2.4.7 ベースである．NEC Linux にはメモリアフィニティ（first-touch 方式により，ローカルの物理ページを割り当てる）およびプロセッサアフィニティ（あるプロセスが実行されることのできる CPU を指定する．プロセスが他の CPU に移動することによるデメリットを解消）機能が実装されており，以下の評価では特に記述がない限りこれらを有効にした状態で測定した．

メモリアフィニティ機能を無効にするとつねに 1 番目のセル上の主記憶から順番にメモリ領域が確保されるため，メモリアクセスがそこに集中し，著しくパフォーマンスが下がる．

またコンパイラには Intel Compiler for Linux 6.0 の `ecc`，コンパイラオプションは -O3 -openmp を用い，並列化はすべて OpenMP[4]を用いて行った．なお，AzusA 上の 4 個のセル上には，それぞれ 2 個の Itanium プロセッサと 128 MB × 4 の PC100 SDRAM が搭載されている．

(1) 2 次元配列を転置
(2) N_1 組の N_2 点 FFT
(3) ひねり係数の乗算
(4) 2 次元配列の転置
(5) N_2 組の N_1 点 FFT
(6) 2 次元配列の転置

図 3 six-step FFT アルゴリズム
Fig. 3 six-step FFT algorithm.

3. FFT アルゴリズム

Fast Fourier Transform（FFT）[5]は離散フーリエ変換（DFT: Discrete Fourier Transform）の計算を高速に実行するアルゴリズムである．離散フーリエ変換は以下のように定義される．

$$Y(t) = \sum_{k=0}^{N-1} X(k)\omega_N^{kt} \qquad t = \{0, 1, \ldots, N-1\}$$

ここで $\omega_N = e^{-2\pi i/N}, i = \sqrt{-1}$ である．

代表的な並列 FFT アルゴリズムである six-step FFT[6]は**図 3** のような 6 つのステップで構成されるアルゴリズムである．ただし入力データのサイズ $N = N_1 \times N_2$ とする．

長さ N の入力データを $N_1 \times N_2$ の 2 次元配列として扱い，その配列に対して行列のように転置を行う．この操作は次に FFT の計算を行うデータが連続したアドレスに並ぶようにするためである．

FFT の計算はメモリアクセスが多い部類に入るが，以下に紹介するような CPU のキャッシュを効率良く利用する手法[7]を用いることで主記憶へのアクセスは低減することが可能である．

N_1 点 FFT および N_2 点 FFT で使うデータがプロセッサのキャッシュメモリに収まるサイズであるような場合，six-step FFT アルゴリズムのように明示的に転置せず，1 列ずつキャッシュへコピーし，キャッシュ上で計算した後に主記憶へ書き戻すという方法をとることによって主記憶へアクセスする回数を減らすような改良を行う．これにより READ および WRITE が

```
for (i = 0; i < N1; i++) {
  /*  i 列目をキャッシュへコピー  */
  LOAD1(CACHE,i);
  IN_CACHE_FFT(CACHE);
  /*  キャッシュから i 列目へコピー */
  STORE1(CACHE,i);
}

for (i = 0; i < N2; i++) {
  /*  i 行目をキャッシュへコピー  */
  LOAD2(CACHE,i);
  IN_CACHE_FFT(CACHE);
  /*  キャッシュから i 列目へコピー  */
  STORE2(CACHE,i);
}
```

図 4 キャッシュを有効利用する six-step FFT
Fig. 4 six-step FFT with cade memory.

それぞれ 2 回ずつにまで減少することができる．

このアルゴリズムの具体的なプログラムは**図 4** のようになる．LOAD1()，LOAD2() で主記憶からキャッシュへのコピーを，STORE1()，STORE2() でキャッシュから主記憶へのコピーを行っている．

N が大きすぎるために N_1 点 FFT および N_2 点 FFT がキャッシュに収まらないような場合，$N = N_1 \times N_2 \times N_3 \ldots \times N_j$ とより多くの因数の積に分解し，データを j 次元配列にみたてて同様に $3j$-step FFT を用いればよい[8]．

3.1 in-place アルゴリズムへの変更

図 4 のアルゴリズムで用いられる IN_CACHE_FFT() に注目する．この部分はプロセッサ内部のキャッシュ上で実行されるため容量的な制約がある．ここで用いる FFT アルゴリズムを Cooley-Tukey FFT[5),9)] 等のデータを上書きする in-place アルゴリズムにすることによって，Stockham FFT アルゴリズム[9)]の半分の容量で計算を実行できるため，あるキャッシュ容量で計算できるサイズが大きくなる．また，L3 キャッシュから L2 キャッシュへとより上位のキャッシュに収まる可能性が大きくなるという利点がある．そこで図 4 のアルゴリズムに改良を加えた，in-place アルゴリズムに対応するアルゴリズムを提案する．

今回用いた Itanium の場合 L3 キャッシュは 2 MB，L2 キャッシュは 96 KB であり，2^{12} 点 FFT を計算する場合 in-place ならば L2 キャッシュに収まるが Stockham FFT の場合は 128 KB となり L3 キャッシュで計算しなければならない．L2 キャッシュと L3 キャッシュでは帯域，レイテンシともに差があり，キャッシュ内の FFT の計算時間にも大きな差が生じる．

表 2 2^{24} 点 FFT の各処理の実行時間の比較（ms）
Table 2 Elapsed time of each part in 2^{24}-point FFT (ms).

	Cooley	stockham
Comp.	209	396
LOAD1	93	99
STORE1	56	63
LOAD2	92	78
STORE2	134	130
Total.	627	799

in-place アルゴリズムを使用するには bit-reverse 処理と呼ばれるデータの格納順序を変える操作が必要となる．そこで図 4 のアルゴリズムの LOAD1() および LOAD2() の中で bit-reverse 処理をすることを考える．これらのルーチンでは主記憶から読み込みを行っており，プロセッサの実行パイプラインは大部分の時間はデータ待ちでストールしている．この無駄になっている時間を利用して bit-reverse 計算すればオーバヘッドなしでキャッシュ内の FFT を in-place アルゴリズムへ変更することができる．なお，n ビットの bit-reverse 計算は Itanium では (n + 2) サイクルで計算でき，メモリアクセス命令と同時に実行できる．

また STORE1() 内でひねり係数の乗算，STORE2() で $1/N$ の乗算も行われている．

表 2 はキャッシュ内の FFT の計算に In-place アルゴリズムとして Cooley-Tukey アルゴリズムを用いた場合と，In-place でない Stockham アルゴリズムを用いた場合を比較したものである．

LOAD2 では少し bit-reverse 処理のオーバヘッドが出ているが，LOAD1 では bit-reverse 処理の隠蔽ができていることが分かる．キャッシュ内の FFT の計算部分では必要なメモリが少ない Cooley-Tukey FFT の方が高速に実行されている．

3.2 Itanium 向けの最適化

キャッシュ上の FFT の計算（IN_CACHE_FFT()）には積和演算命令に適した radix-4 FFT Kernel[10)]を使用し，必要な三角関数のテーブルはあらかじめ作成してある．今回用いた IN_CACHE_FFT() のコードを Intel Math Kernel Library（MKL）5.1 に含まれる FFT ライブラリ関数と性能比較してみたところ，長さ 2^{10}，2^{12} では IN_CACHE_FFT() が 10%ほど高速であったが，データの格納方式等の条件が異なるため単純に比較することはできない．

radix-4 FFT Kernel は演算数が多く，ソフトウェアパイプラインのステージ数も多くなるためプロローグ/エピローグ部分のオーバヘッドも大きい．このオーバヘッドを低減させるため，再内側の二重ループを一

重ループにする等の最適化を施している．また，2 次元配列の各行には L2 キャッシュのブロックサイズに合わせて 64 byte か 128 byte☆の使わない領域を加えることで，縦方向にアクセスするときにキャッシュラインの競合等が起こらないようにしており，さらに 4 列同時にアクセスすることによって 64 byte ブロック単位で起こるキャッシュと主記憶間のデータ転送レートを最大限に利用している．プログラムの並列化は OpenMP を用いて図 4 の 2 つの for ループを並列化した．粗粒度の並列化であるため同期等のオーバヘッドは無視できる．なお，スケジューリング方式には一番高い性能を出した static を使用している．

LOAD1()，LOAD2() では主記憶からデータを読み込むため非常にレイテンシが大きく，スループットを上げるために，rotation register を最大限利用してつねに 80 個の倍精度浮動小数点数の load 要求を先行して発行している．これに対して STORE1()，STORE2() ではキャッシュから読み込むためレイテンシは小さいため，ひねり係数等の乗算はこちらに組み込んでいる．なお，主記憶アクセスの大きなレイテンシに対応するため，これらの関数にはアセンブリ言語で記述したコードを用いている．

4. メモリ性能の予備評価

FFT の計算速度には主記憶の性能が大きく影響する．計算機のメモリ帯域幅によって最適な FFT の実装が異なる．このためまず AzusA の実効メモリ帯域幅を計測した．メモリアクセスの方法によって計測される帯域幅は変化するが，今回は倍精度浮動小数点数の 1 次元配列に対して各要素の自乗和を求めるプログラムを用いた．

表 3 はその計測結果である．各スレッドあたりのデータは 128 MB 固定で，合計（128 × スレッド数）MB の double 型の配列を用いた．並列化には OpenMP を使用，並列に配列の初期化を行うことで各セルのローカルの主記憶に確保されている．スレッド数を 1〜8 の範囲で変更し，各スレッドをなるべく多くのセルへ分散するように配置し，最後まで同じ CPU で実行されるように各スレッドを CPU に固定して計測した．スレッド数 1 から 4 では各セルに最大でも 1 スレッドしか割り当てていないため，各セルの資源をほぼ独占できる状態にある．しかしセル間のバスや Tag メモリの飽和により 2.5 GB/s ほどまでしか上がらない．

スレッド数 8 ではスレッド数 4 のときと比較して各

☆ Itanium では 64 byte，Itanium2 では 128 byte

表 3 実効メモリ帯域幅（なるべく各セルに分散して配置）
Table 3 Effective memory bandwidth (with maximum number of cells).

スレッド	1	2	3	4	8
帯域幅（MB/s）	1082	2166	2588	2564	2468

表 4 実効メモリ帯域幅（セルを 1 つずつ埋めていく配置）
Table 4 Effective memory bandwidth (with minimum number of cells).

スレッド	1	2	4	6	8
帯域幅（MB/s）	1082	792	1577	2361	2468

セルで使用する CPU 数が増えているだけであるが，すでにメモリ帯域幅が限界に達しているため全然性能が上がらない．

表 4 はプロセッサ割当てを変更したものである．スレッド数を増やしたときに 1 番目のセルから順に埋めるように割り当てている．割当て方を変更するとまったく異なる結果となったが，各セルに分散して割り当てるようにした場合の方が高い性能が出ている．いずれの配置でも全スレッドの合計帯域幅は 1 スレッド時の 2.4 倍以下とスケーラビリティーが乏しい．また，絶対性能としても 2.4 GB/s という値はプロセッサの 1 サイクルあたり約 3 byte であり，8 CPU 分の演算性能（1 サイクルあたり 16 演算）に対して決して十分とはいえない．このため AzusA において FFT の計算を高速化するにはキャッシュを有効利用することが重要である．

なお，zaxpy 等 READ だけでなく WRITE を含むメモリアクセスを数種類試してみたところ，いずれも READ と WRITE の帯域幅を合計した値は先ほどの値の 90%から同等になった．

5. データの分散配置方式

six-step 系アルゴリズムで用いる 2 次元配列データを cc-NUMA の分散共有メモリへ分散配置する方法は**図 5** のような 1 次分割を用いた．cc-NUMA システムでは連続する仮想ページを異なるセルに割り当てることができるが，物理ページ単位（8 kB）より細かい分割は不可能であるため，現実的にはこのような 1 次元分割以外の割り当て方をすることは難しい．

FFT の計算では，six-step FFT 系のアルゴリズムを用いる場合 2 次元配列の横方向（行方向）への連続アクセスと，縦方向（列方向）の非連続アクセスが必要であり，このような 1 次元分割では縦方向の非連続，とびとびのアクセスでセル間通信が起こる．なお，縦方向のアクセスにおいてキャッシュラインの競合を避けるため，各行の間に 64 byte の padding を加えて

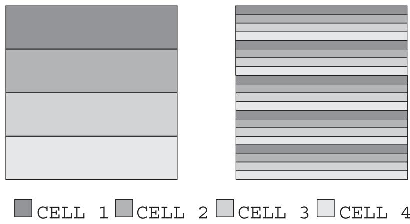

図 5 2 次元配列データの各セルへの分配方式．
左が block 分割で右が cyclic 分割

Fig. 5 Methods to distribute 2-dimensional arraydata. Block (left) and cyclic (right).

表 5 データ配置と FFT の各処理の実行時間（ms）

Table 5 Elapsed time of each part for different data placement.

	default	block	cyclic
Comp.	318	264	264
LOAD1	271	130	126
STORE1	147	66	69
LOAD2	201	90	106
STORE2	435	121	137
Total.	1372	671	702

いる．

表 5 はデータ配置を変えることによって実際に FFT の計算時間にどのような差が出るか測定したものである．分割方法には

（1） 分割なし（default）

（2） 縦方向に block 分割

（3） 縦方向に cyclic 分割

の 3 つを用いた．

8 スレッドで 2^{24} 点 FFT の計算を行ったもので，数値は図 4 のアルゴリズム内の各関数（Comp. は IN_CACHE_FFT()）の実行に要した時間（ミリ秒）で，8 スレッドそれぞれで計測した時間の平均値を用いている．

default はセル 1 に集中して割り当てられるため性能が低く，スレッド間で実行時間に不均等も生じている．block 分割と cyclic 分割を比較すると block 分割の方が合計の実行時間は短い．これは LOAD2 のタスク割当て方式が block 分割であり，データ分割方式と一致するためである．cyclic 分割のタスク割当て方式には cyclic 方式を使用するべきであるが，LOAD2 では主記憶の転送効率の問題で 4 行ずつアクセスしており cyclic にすることは不可能であるため，block 分割方式のタスク割当てを使用している．LOAD1 だけは cyclic 分割の方が速くなったが，cyclic 分割では各セルの主記憶に交互にアクセスするため，同じセルの主記憶にアクセスし続けるより速くなったものと考えられる．

表 6 2^{16} 点 FFT の計算時間（ms）と Mflops 値

Table 6 Elapsed time (ms) and performance (Mflops) in 2^{16}-point FFT.

	Total		Memory	Comp.	
#PE	Time	Mflops	Time	Time	Mflops
1	10.77	486	7.04	3.73	1405
2	5.85	895	3.69	2.16	2425
4	3.89	1345	2.67	1.22	4297
8	3.14	1667	2.46	0.68	7642

表 7 2^{20} 点 FFT の計算時間（ms）と Mflops 値

Table 7 Elapsed time (ms) and performance (Mflops) in 2^{20}-point FFT.

	Total		Memory	Comp.	
#PE	Time	Mflops	Time	Time	Mflops
1	218.8	479	149.7	69.1	1517
2	102.0	1027	67.0	35.0	2991
4	53.2	1970	35.4	17.8	5874
8	35.7	2933	26.5	9.2	11298

表 8 2^{24} 点 FFT の計算時間（ms）と Mflops 値

Table 8 Elapsed time (ms) and performance (Mflops) in 2^{24}-point FFT.

	Total		Memory	Comp.	
#PE	Time	Mflops	Time	Time	Mflops
1	4045	497	2422	1623	1240
2	2232	901	1320	862	2334
4	1133	1775	709	424	4740
8	644	3123	432	212	9458

以上の結果より AzusA で FFT を計算する場合，データを 2 次元配列とすると縦方向に一次元ブロック分割する方法が適していると判断できる．以降ではこの配置方法を用いた．

6. 性能評価

表 6, **表 7**, **表 8** は PE 数を変更したときの並列 FFT の性能を測定したものである．全体の実行時間と，主記憶アクセス時間（Memory）およびキャッシュ上での計算時間（Comp.）をそれぞれ測定した．Memory は図 4 のアルゴリズム中の LOAD1()，STORE1()，LOAD2()，STORE2() の実行時間の合計で，Comp. は IN_CACHE_FFT() の実行時間である．それぞれ各スレッドでの測定時間の平均値をとっている．浮動小数点数演算性能（Mflops）に関しては FFT に要する浮動小数点数演算量を $5N \log_2 N$ として計算した．

データは倍精度浮動小数点数の複素数で，長さはそれぞれ $N = 2^{16}$，2^{20}，2^{24}．時間の単位はミリ秒である．

まず $N = 2^{16}$ の場合についてはデータサイズが合計 1 MB と小さく，1 CPU のキャッシュに収まってしまう．1 スレッドと 2 スレッドの結果を比較してみて

も並列に計算するべきではない長さであることが分かり，細かく分割したためのオーバヘッドが現れている．CPU の割当て方法を変更して 2 つのスレッドを同じセルに割り当てた場合もわずか 5%ほど性能が上がる程度であった．$N = 2^{16}$ の実験結果に関してはこれ以上特に解析しない．

Comp. に関しては，キャッシュにあるだけで計算できるため当然 PE 数が増えても $N = 2^{24}$ の場合 8 PE で 1 PE の約 7.6 倍と 8 倍には若干及ばないが，高いスケーラビリティが容易に得られている．特に $N = 2^{24}$ の場合には L3 キャッシュへのアクセスも起こっているが，in-place アルゴリズムを使用しているために大部分は L2 キャッシュにヒットしており，性能の低下が小さい．

これに対して PE 数が増加するにつれて主記憶にアクセスする時間はゆるやかに減少している．並列 FFT ではリモートセルの主記憶に対してのロードとストアが 1 回ずつ起こるため，この部分ではスケーラビリティを得るのが難しい．また，特に 4 PE と 8 PE の差は少ないが，これは CPU 割当ての方法とシステム全体の主記憶アクセス帯域の限界によるものである．4 PE の場合には各セルから 1 CPU ずつを使用しているため各セルの CPU バス，メモリを効果的に使えているのに対して，8 PE の場合はそのまま各セル 2 CPU ずつとなるため性能が上がりにくい．

主記憶へのアクセスにかかる時間が FFT の計算時間全体の 59% から 67% 程度と大部分を占めているため，4 PE で約 3.6 倍という並列化効率は 1 PE の場合すべてローカルの主記憶であることを考慮すれば妥当な数値であるが，8 PE では約 6.3 倍とあまり性能が出ない結果となった．

メモリ性能測定時と異なり，今回用いたような FFT の計算では縦方向の非連続アクセスや，リモートセルの主記憶へのアクセスも起こるため，高速なメモリアクセスができていない．2^{24} サイズでは主記憶のデータ転送量は 256 MB の READ 2 回，WRITE 2 回で合計 1 GB である．全実行時間で 1 GB 転送すると考えて表 8 の結果から計算すると，1 PE で 253 MB/s，8 PE で 1.59 GB/s（PE あたり 198 MB/s）となり，メモリ帯域幅としてはまだ余裕があることが分かる．

原因として考えられることとしてはまず，計算を行っている時間とメモリアクセスをしている時間が完全に分離しているため全体のメモリアクセス率は低いということが考えられる．このように明確に分離せず，キャッシュ上で計算している間にも主記憶へのアクセスを並行して行うことで計算時間とメモリアクセス時間をマージし，いわゆる通信時間の隠蔽のような手法を用いることで合計の実行時間を短くできる可能性はある（実際にはメモリアクセス時間の方が長いので，隠蔽されるのは計算時間である）．ただし，このような手法はアーキテクチャの制約が厳しく，特に IN_CACHE_FFT() でもロード/ストア命令は実行されているため，これらの命令の実行に影響を与えずにうまく高速化を実現するのは難しい．マージに失敗すれば競合が起こり，余計に時間がかかってしまう．いくらか試みたものの現時点ではまだ成功していない．

表 9 Itanium と Itanium2 で FFT の性能比較．それぞれ 1 CPU で測定した．単位は Mflops

Table 9 Performance (Mflops) of Itanium and Itanium 2. 1 CPU is used.

	$N = 2^{16}$	$N = 2^{20}$
Itanium 733 MHz	555.1	471.5
Itanium2 900 MHz	771.2	861.8

またもう 1 つの考えられる原因としては，キャッシュ内で FFT を計算している際に store 命令によってキャッシュに書き込まれたデータが一部主記憶へも転送されている可能性があることがあげられる．非連続アドレスへのアクセスや，リモートセルの主記憶へのアクセスが起こることも関係している．

浮動小数点数演算のピーク性能は $2933 Mflops \times PE$ であるが，これに対して表 8 では 1 PE で 16.9%，4 PE で 15.1%，8 PE で 13.3%というピーク性能比となった．この数値は大体他のマシンでのピーク性能比と同程度である．

先日 Itanium の次の世代の IA-64 プロセッサである Itanium2（コードネームは McKinley）が発表された．基本的な機能は Itanium と変わらないが様々な改良が行われ，メモリアクセス命令等を処理する M-Unit が 2 個から 4 個に増設されており，CPU バスの帯域幅も 2.1 GB/s から 6.4 GB/s へと上がっている等，特にメモリまわりの性能が強化されている．これらは FFT の計算を行う場合の性能にも大きく影響する要素であり，**表 9** のように実際 Itanium に比べてかなり良い測定結果が出てきている．CPU バスのデータ転送レートが大幅に上がったのが主な原因と考えられる．

7. ま と め

本稿では，NEC の分散共有メモリ型並列計算機である AzusA を用いて，共有メモリアーキテクチャ向きの並列 FFT アルゴリズムの分散共有メモリアーキテクチャ上への実装方式について検討し，また AzusA の主要な機能に関して性能評価を行った．SMP アー

キテクチャの場合と同様にキャッシュを有効利用する手法が有効であったが，本研究で提案した in-cache FFT アルゴリズムに対応した手法によって，キャッシュ内でより大きなサイズの FFT の計算をする場合にも性能の低下を小さくすることができた．さらに分散共有メモリであることを考慮したデータ配置を行うことが高い性能を出すには必要不可欠であることが分かった．Itanium の高い CPU 性能を活かすための適切な FFT カーネルの実装と組み合わせることでより高い性能を引き出すことが可能となる．メモリアクセスに全実行時間の半分以上を占められているにもかかわらず，2^{24} 点 FFT の計算を 8 PE で行った場合では 3.123 GFlops（ピーク性能の 13.3%）という高い性能を出すことができた．Itanium2 ではメモリ関連の機能が強化されており，さらに高い性能が出ることが期待される．

謝辞 本研究を進めるにあたり，日本電気株式会社より様々なサポートおよびアドバイスをいただきました．感謝いたします．なお，本研究の一部は，科学研究費補助金基盤研究（B）13480080，特定領域研究（C）（2）14019030，および科学技術振興事業団戦略的創造研究推進事業によるものである．

参 考 文 献

1) Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A. and Hennessy, J.: The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor, *Proc. 17th Annual International Symposium on Computer Architecture*, pp.148–159 (1990).
2) Intel Itanium Processor. http://developer.intel.com/design/itanium/
3) Aono, F. and Kimura, M.: The AzusA 16-Way Itanium Server, *IEEE Micro*, Vol.20, No.5, pp.54–60 (2000).
4) OpenMP: Simple, Portable, Scalable SMP Programming. http://www.openmp.org/
5) Cooley, J.W. and Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series, *Math. Comput.*, Vol.19, pp.297–301 (1965).
6) Van Loan, C.: *Computational Frameworks for the Fast Fourier Transform*, SIAM Press, Philadelphia, PA (1992).
7) 高橋大介：共有メモリ型並列計算機における並列一次元 FFT のブロックアルゴリズム，2001 年並列処理シンポジウム論文集，pp.359–366 (2001).
8) 高橋大介，朴 泰祐，佐藤三久：PC クラスタにおける並列一次元 FFT のブロックアルゴリズム，2002 年並列処理シンポジウム論文集，pp.55–62 (2002).
9) Swarztrauber, P.N.: Multiprocessor FFTs, *Parallel Computing*, Vol.5, pp.197–210 (1987).
10) Goedecker, S.: Fast Radix 2, 3, 4 and 5 Kernels for Fast Fourier Transformations on Computers with Overlapping Multiply-Add Instructions, *SIAM J. Sci. Comput.*, Vol.18, pp.1605–1611 (1997).

(平成 14 年 9 月 23 日受付)
(平成 15 年 1 月 12 日採録)

額田　彰（学生会員）

1976 年生．1999 年東京大学理学部情報科学科卒業．2001 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同年より東京大学大学院情報理工学系研究科コンピュータ科学専攻博士課程在学中．特に高速フーリエ変換に興味を持つ．

西田　晃（正会員）

1970 年生．1995 年東京大学理学部情報科学科卒業．1998 年東京大学大学院理学系研究科情報科学専攻博士課程修了．理学博士．同年より東京大学大学院理学系研究科情報科学専攻助手．2002 年より科学技術振興事業団戦略的創造研究推進事業「シミュレーション技術の革新と実用化基盤の構築」領域研究代表者を兼務．反復解法，特に大規模固有値解法と並列数値処理の研究に従事．ACM，IEEE，SIAM，日本応用数理学会，日本ソフトウェア科学会各会員．

小柳　義夫（正会員）

1943 年生．1966 年東京大学理学部物理学科卒業．1971 年東京大学大学院理学系研究科物理学専門課程修了，理学博士．同年東京大学助手．高エネルギー物理学研究所理論部門助手，筑波大学電子情報工学系講師，助教授，教授を経て，1991 年東京大学理学部情報科学科教授．並列処理，数値解析，計算物理学に関する研究に従事．特に，偏微分方程式の高速並列解法，最小二乗法の数値計算，乱数やモンテカルロ法に興味を持つ．物理学会，日本統計学会，応用統計学会，計算機統計学会，応用数理学会等各会員．