ハードウェア分散共有メモリを用いた疎行列アルゴリズムの細粒度並列処理

西田 晃[†] 額田 彰[†] 小柳義夫[†]

本研究では, コモディティハードウェアによって構成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, ユーザが共有メモリアーキテクチャの下位構造を意識することなく高性能な科学技術演算を行なうことのできる計算環境の構築を目指している. 本稿では, そのための予備評価として NEC Itanium ccNUMA サーバ上に実装された Linux システムを利用し, 疎行列アルゴリズムの細粒度な実装方式とその最適化手法について, 実機上での評価結果をもとに考察を行った. SGI Origin 2000 との比較の結果, コモディティ分散共有メモリの構築に際しては, メモリ帯域幅の確保に注意する必要のあることが明らかとなった.

Fine Grain Parallel Implementation of Sparse Matrix Algorithms on Hardware Distributed Shared Memory Architectures

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

In this study, we target a commodity NUMA computing environment which enables users to realize high performance scalable scientific computing without thinking the details of its architecture. As a preliminary evaluation, we discuss in this paper the implementation of sparse matrix algorithms and its optimization on Linux environment implemented on an NEC Itanium based ccNUMA server. The comparative study with SGI Origin 2000 shows that we have to be aware of sustained memory bandwidth to construct high performance commodity NUMA environments.

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. 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不均等になるものの, プロセッサの拡張性に関する物理的制約はほとんどなく, 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[6),8),10)].

一方で, コモディティプロセッサの高速化に伴い, 高速の PC をネットワークで結合したクラスタ技術が, 大規模科学技術計算においても実用的な選択肢のひとつとなってきている. しかしながら, クラスタ上でのメッセージ通信による並列処理には, 明示的なデータ分割の必要性や通信遅延などの制約がある. 現在, このような安価なノードを組み合わせて, 大規模な分散共有メモリ環境を構築するためのチップセット等の開発が Intel, IBM 等の主要ベンダにより進められている[11)].

本研究では, 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スケーラブルな性能向上を得る上で障害となるオペレーティングシステム上のボトルネックを解消し, 高レベルな最適化を実現することにより, ユーザがアーキテクチャの下位構造を意識することなく高性能な科学技術演算を行なうことのできる計算環境の構築を目指している. 本稿では, そのための予備評価として, 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, 疎行列アルゴリズムの細粒度な実装方式とその最適化手法について, 実機上での評価結果をもとに考察を行った.

2. 背景

大規模疎行列の反復解法においては, 疎な成分を持つ行列とベクトル間の演算が計算量の大部分を占めるため, この演算の効率的な処理が必要不可欠である. これらは Dongarra らによって開発された BLAS (Basic

† 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, the University of Tokyo

Linear Algebra Subprograms)[9] を用いて実装することが可能であるが, 行列 - ベクトル間演算においては, 一般に演算量は $O(n^2)$ であり, 十分にキャッシュメモリを活用するのは難しい. したがって, 疎行列アルゴリズムの並列化を行う場合, BLAS レベルでの並列化が最適な選択肢であるかどうかを決定するためには, 十分な評価が必要である.

BLAS の並列実装に関しては, 1) ScaLAPACK[4] のように, MPI による BLAS と等価な並列化ライブラリを構築するもの, と, 2) ATLAS[12),13)] のように, pthreads により並列化を行うものがあり, 1), 2) とも一部の BLAS ルーチンについては, 効率的な並列実装が実現している. ただし, PBLAS は分散メモリアーキテクチャを主な対象としており, また, ATLAS の Level 3 BLAS ルーチンに関する対称型マルチプロセッサ向けの並列化は, 本アルゴリズムのような疎行列解法への適用を想定したものではない.

しかしながら, メモリアクセス遅延の小さい共有メモリアーキテクチャ上においては, ループレベルでの並列化によって, Level 1, 2 BLAS 演算についても効率的に性能向上を達成できるものと期待される. 本稿では, 大規模疎行列アルゴリズムの BLAS レベルでの並列化について, 評価の結果を報告するとともに, 疎行列用ライブラリの構築の際の注意点について考察する.

3. 実装と性能評価

3.1 評価環境

現在, Intel 社及び Hewlett Packard 社により開発の進められている IA-64 アーキテクチャは, 過去に開発された Intel アーキテクチャとの互換性を維持している点で, 他の高性能プロセッサにはない特色を持っている. 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 512CPU までメモリを共有することができる. 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ひとつとして, NEC の AzusA Itanium サーバを挙げることができる. 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.

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 個のプロセッサ及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によるコヒーレンシ実装で接続することにより, 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 1,2 に AzusA のシステム及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 2.1GB/s, 4.2GB/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 4.2GB/s, 16.8GB/s である.

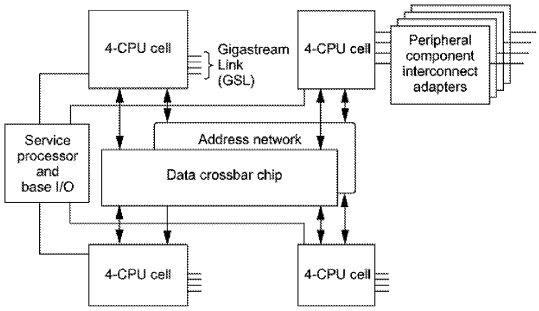

図 1 AzusA system block diagram.

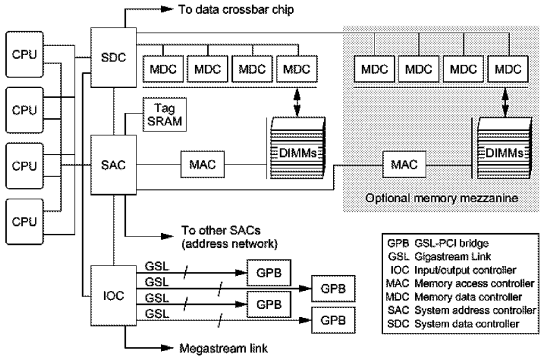

図 2 AzusA chip set components.

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 チップセット及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 200ns 以下, リモートセルからの読み出しに対して 300ns 以下のメモリレイテンシを実現している.

4. 実　　装

分散共有メモリアーキテクチャ上において, オペレーティングシステムレベルでのスケーラビリティを確保するための手法としては, 以下のような手法が提案されている.

ノードアフィニティ
プロセスの移動を指定したセル内に限定する機能[7].

メモリアフィニティ
物理的なメモリ配置情報に基づいて, プロセスのメモリ要求時にセルローカルのメモリを優先的に割り当てる機能.

プロセッサアフィニティ
プロセスを指定したプロセッサに固定する機能.

ページマイグレーション
あるメモリページに対して特定のリモートノードからのアクセスが集中する場合, OS がこれを検知してページをリモートノードにコピーする機能. オーバヘッドに注意する必要がある.

これらの機能により, リモートノードへのメモリアクセスが減少し, データの局所性が向上するとともに, CPU 間のプロセス移動によるキャッシュミスの発生を抑えることができる.

分散共有メモリアーキテクチャ上で高い計算性能を得るためには, 計算アルゴリズムにおいてもさまざまなレベルでの最適化が必要である. キャッシュレベルでの最適化に関しては, 上述の ATLAS などの研究があるが, 現段階では, 並列アーキテクチャレベルでの最適化については考慮されていない. このようなシステムを研究する上で, 仕様が完全に公開され, 無償でソースコードの入手及び自由な改変が可能なオペレーティングシステム及び各種ライブラリの開発が, ネットワークの普及に伴って一般的になりつつある. そのようなオペレーティングシステムのひとつとして, Linux を挙げることができるが, Linux において実現されているスケジューリング機能は、現時点では比較的小規模なシステム向けに設計されており, 本研究で対象とする大規模な共有メモリアーキテクチャ上で使用するには, 性能上問題がある. 以上の背景をもとに, 主要ベンダを中心に, Linux においてスケーラビリティを向上させるための研究が進められている[1]. 本研究では, AzusA 上に実装された Linux オペレーティングシステムを用いて, NUMA 上の OS カーネルの機能と性能を評価した.

評価には, NEC AzusA (733MHz Intel Itanium Processor × 8, 32KB L1 cache (16KB data/16KB instruction), 96KB L2 cache, 2MB L3 cache, AzusA chipset, 2GB main memory), また比較対象として, 産業技術総合研究所の SGI Origin 2000 (400MHz R12000 × 32, 64KB L1 cache (32KB data/32KB instruction), 8MB L2 cache) を用いた. 前者の OS は Red Hat 7.1 ベースの NEC Linux R1.2, 後者は IRIX 6.5 である. NEC Linux にはメモリアフィニティ及びプロセッサアフィニティ機能が実装されており, 以下の評価ではメモリアフィニティを有効にした状態で測定した. またコンパイラにはそれぞれ Intel Compiler for Linux 6.0 の `efc`, 及び MIPSpro 7.3.1.2m の `f90` を用いた. 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 × 4MB の PC100 SDRAM が搭載されている.

4.1 EPCC OpenMP Microbenchmark

OpenMP の実行モデルでは, プログラムの実行はマスタスレッドと呼ばれる単一プロセスとして開始される. マスタスレッドは通常のステートメントを逐次実行し, PARALLEL と END PARALLEL 指示文の対で構成される並列構造が現れると, 1 つ以上のスレッドからなるチームを生成し, チームのメンバのそれぞれについてデータ環境の設定を行なう. 並列構造内のステートメントは, チーム内の各スレッドによって並列に実行され, 並列構造の終了時点でチーム内のスレッドは同期し, マスタースレッドは更新されたデータを用いて計算を続ける.

共有メモリアーキテクチャはメモリアクセス遅延が小さい反面, 分散メモリアーキテクチャと比較してキャッシュミスの発生が多く, また, 同一のキャッシュラインへの書き込みが発生した場合には著しく性能が低下する場合がある. しかしながら, この点についても留意して実装を行うことにより, 高い性能を確保することが可能である.

ここでは, OpenMP の使用とハードウェア性能との関係について調べるため, EPCC OpenMP Microbenchmark[5] による評価を行った. Microbenchmark は, エジンバラ大学で開発された OpenMP の同期・スケジューリングオーバヘッド測定のためのベンチマークであり, 内部に dummy 関数を含むループについて, OpenMP 指示文を追加した場合に生じる遅延を測定する. すなわち, バリア同期, 排他制御についてはそれぞれ

```
!$OMP PARALLEL
      do j=1, innerreps
!$OMP DO
      do i=1,omp_get_num_threads()
         call delay(delaylength)
      end do
!$OMP END DO
      end do
!$OMP END PARALLEL


!$OMP PARALLEL
         do j=1, innerreps/omp_get_num_threads()
!$OMP CRITICAL
            call delay(delaylength)
```

```
!$OMP END CRITICAL
      end do
!$OMP END PARALLEL
```

のように, PARALLEL directive が無視できる回数のループを実行し, 逐次に実行した場合との比較によりオーバヘッドを測定する. またループスケジューリングについては

```
!$OMP PARALLEL
      do j=1, innerreps
!$OMP DO SCHEDULE(schedtype,chunksize)
         do i=1,itersperthr*omp_get_num_threads()
            call delay(delaylength)
         end do
      end do
!$OMP END PARALLEL
```

としてスケジューリング方式とデータの分割サイズを指定し, 結果を測定する.

ここでは AzusA と Origin 2000 について, それぞれのオーバヘッドを測定した. 図 3, 4 に AzusA 上, また図 5, 6 に Origin 2000 上での同期オーバヘッドを示す. 時間測定にはナノ秒オーダの実行時間を計測することのできる関数 `clock_gettime()` を用いた. 縦軸は実行時間をクロックサイクル数に換算した値である. Intel Compiler にはコンパイラオプションとして `-O3 -zero -ip -nodps -w -openmp` を, また MIPSpro には `-Ofast -mp` を用いた.

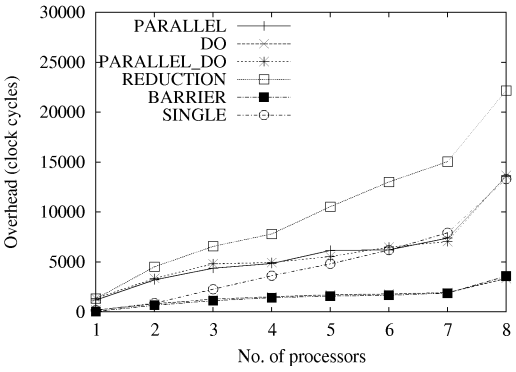

図 3 Synchronization overheads on NEC AzusA.

この結果から分かるように, バリア同期については, 類似の機能を持つ `BARRIER` と `DO`, `PARALLEL` と `PARALLEL DO`, 及び最終的な計算結果を得るために排他制御を行う `REDUCTION` では, AzusA, Origin 2000 ともほぼ同等のオーバヘッドであり, 最後に同期による排他制御を伴う `SINGLE` については, Origin 2000 の値が若干大きい. 排他制御のオーバヘッドは全般的に AzusA

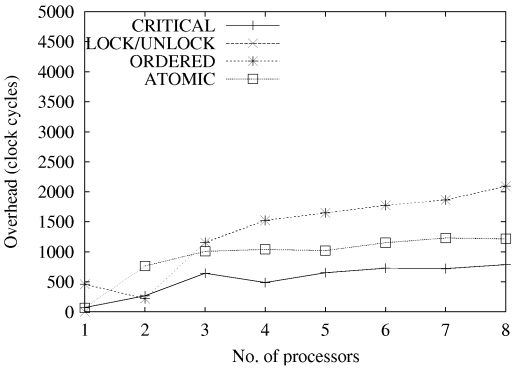

図 4 Mutual exclusion overheads on NEC AzusA.

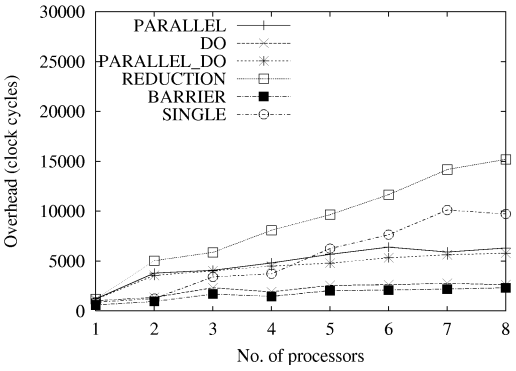

図 5 Synchronization overheads on SGI Origin 2000.

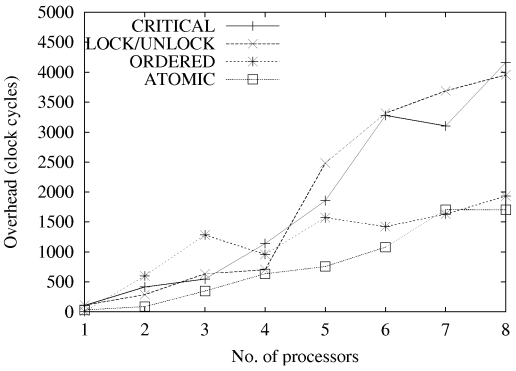

図 6 Mutual exclusion overheads on SGI Origin 2000.

の方が小さく, 特に `CRITICAL`, `LOCK/UNLOCK` で良好な性能を示している. これは AzusA のリモートセルからの読み出しに伴うレイテンシが Origin 2000 に比べて相対的に小さいことによると思われる. なお, AzusA のバリア同期オーバヘッドは 8 スレッドでの実行時に増加しているが, これは全 PE を使用していることによると考えられる. 図 7, 8 に 4 スレッドでのスケジューリングオーバヘッドを示す. chunk サイズにもよるが, AzusA のオーバヘッドは全体的に小さい.

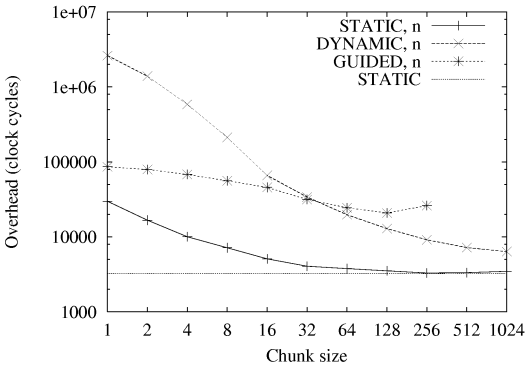

図 7 Scheduling overheads on NEC AzusA.

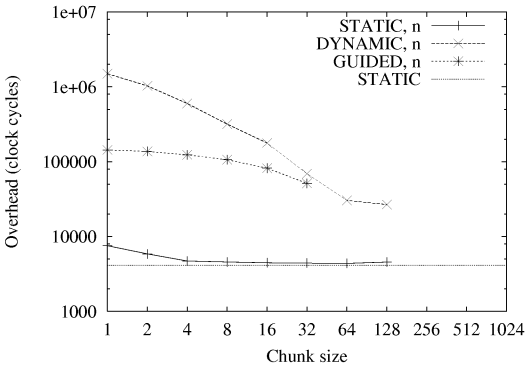

図 8 Scheduling overheads on SGI Origin 2000.

4.2 並列化 BLAS の性能評価

ここでは, 大規模疎行列を対象とした固有値解法である Jacobi-Davidson 法[14] において, 計算量の大部分を占める Level 1,2 BLAS 型の複素演算ルーチン `zgemv`, `zdotc`, `zaxpy`, `dznrm2` を選び, 分散共有メモリシステム上での評価を行った.

なお, `zgemv` は

```
y := alpha*A*x + beta*y,
y := alpha*A'*x + beta*y,
y := alpha*conjg(A')*x + beta*y,
```

型の Level 2 BLAS 演算, また `zdotc`, `zaxpy`, `dznrm2` はそれぞれ

```
z := conjg(x)*y,
y := alpha*x + y,
dznrm2 := sqrt(conjg(x')*x)
```

型の Level 1 BLAS 演算である.

ここでは, それぞれの関数の最外側ループを OpenMP API を用いてブロック化したルーチンを作成し, 並列に処理する. ループ内での変数の私有化によりメモリアクセスの競合を最小限に抑えるとともに, 可能な限り陰的なバリア同期を行なわないよう指定する. ベクトルサイズは 256^2 とし, 各ルーチンとも 10 回ずつ実行した結果を計測した.

同一コードを逐次に実行した場合と, OpenMP 指示文を挿入し, 1 スレッドで実行した場合の性能を表 1に示す. AzusA 上での浮動小数演算性能が Origin 2000 に比べて低く, また並列化コードの性能が逐次実行時に比べて低下しているが, これは Intel Compiler の実装の一部に問題が残っているためであると思われる☆. なお Origin 2000 上での逐次コードの最適化オプションは `-O3` とした.

表 1 Performance (MFLOPS) of serial and single threaded parallel codes.

	AzusA		Origin 2000	
Function	Serial	Parallel	Serial	Parallel
zaxpy	259.1	138.7	334.0	328.2
zdotc	118.1	21.6	285.2	278.7
dznrm2	95.6	93.4	116.5	150.2
zgemv	113.2	52.2	147.0	206.5

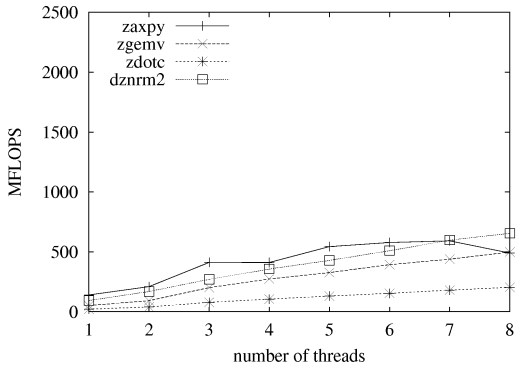

図 9 Parallel BLAS performance on NEC AzusA.

まず, デフォルトのカーネルを使用した AzusA, Origin 2000 上での性能を図 9, 10 に示す. また, プロセッサアフィニティ機能を有効にした場合の AzusA 上での性能は, 図 12 の通りであった☆☆. プロセッサアフィニティ機能を実現するライブラリは kit カーネルモジュールとして NEC Linux に組み込まれており, ライブラリを利用することにより, プロセスを指定したプロセッサに固定することができる. ここではそのために図 11 に示すルーチン☆☆☆を利用し, Fortran で記述されたテストプログラムから呼び出した. ただし図 12 に示したように, このプログラムの場合には特に顕著な効果は見ら

☆ 具体的な評価性能については非開示扱いのため公表できないが, この問題は 7.0 Beta でおおむね解消された. ただし複数スレッド時での性能についてはそれほど改善は見られなかった. これは後述するメモリ帯域幅の問題によるものと考えられる

☆☆ メモリアフィニティ機能についてはスレッド内でメモリを確保する必要があるが, ここでは評価していない.

☆☆☆ 本ルーチンは日本電気株式会社よりご提供頂いた.

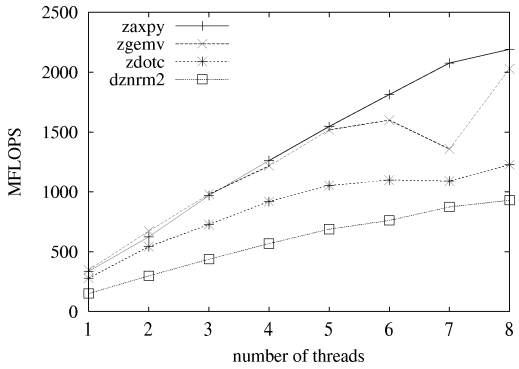

図 10 Parallel BLAS performance on SGI Origin 2000.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include ‘‘/opt/intel/compiler60/ia64/
include/omp.h’’
#include ‘‘/usr/local/include/kit.h’’

void pin_cpu_() {
  int i;
  pid_t *pid;
  unsigned long mask=1;
  pid = (pid_t *)malloc(omp_get_max_threads()
    *sizeof(pid_t));
#pragma omp parallel
  {
    pid[omp_get_thread_num()] = getpid();
  }
  kit_open();
  for (i=0; i<omp_get_max_threads(); i++) {
    if (kit_set_cpumask(pid[i],mask)!=0) {
      printf(‘‘kit_set_cpumask faild for
    thread:%d\n’’,i);
    }
    mask <<= 1;
  }
}
```

図 11 A subroutine to implement processor affinity.

れなかった．またページマイグレーションについては，OS 上に同機能が実装されている Origin 2000 上において評価を行ったが，性能上の差異は見られなかった．これらは主にプログラムの実行時間が短いことによるものと考えられる．

性能上の問題についてより詳しく調べるため，バージニア大学で開発されている STREAM benchmark[2] を用いてそれぞれのシステムのメモリ帯域幅を測定した．STREAM では倍精度浮動小数配列に対して以下の演算を複数回行い，実測値をもとに計算機の性能を評価する．

ここでは OpenMP 版の `stream_d_omp.c` を用い，それぞれのアーキテクチャ上で評価した．なおここではプロセッサアフィニティ機能は使用していない．以下に結

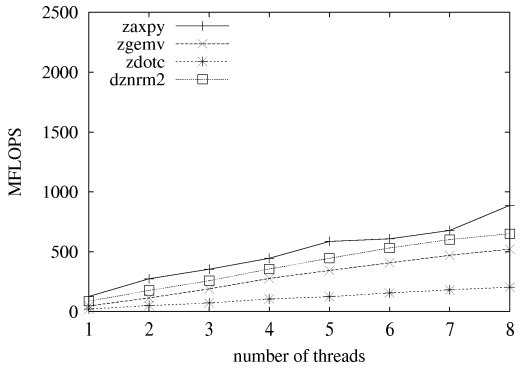

図 12 Parallel BLAS performance on NEC AzusA.

表 2 STREAM benchmark types

Benchmark	Operation	Bytes per iteration
Copy	a[i] = b[i]	16
Scale	a[i] = q * b[i]	16
Add	a[i] = b[i] + c[i]	24
Triad	a[i] = b[i] + q * c[i]	24

果を示す．

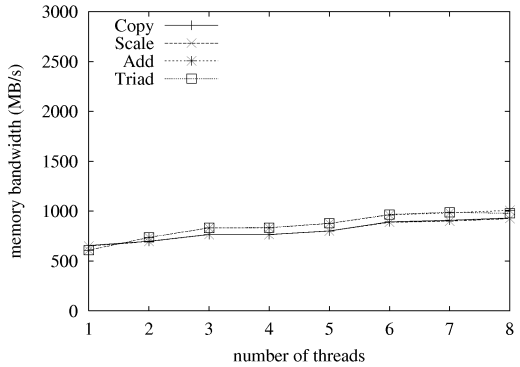

図 13 STREAM benchmark performance (MB/s) with array size 20,000,000 on NEC AzusA (without memory affinity).

図 13, 14から分かるように，AzusA 上でのメモリ帯域幅は，メモリアフィニティ機能を利用した場合 4 スレッドまでは増加するものの，それ以降はアフィニティ機能の効果はなく，約 1GB/s で飽和しているのに対して，図 15 に示すように，Origin 2000 上では 1 スレッド当りの帯域幅は小さいが，スレッド数に比例した性能向上が得られていることが分かる．

AzusA 上で十分な性能が得られていない点については，以下の原因が考えられる．AzusA では，4PE 構成のセル内をフロントサイドバス，またセルカード間をデータクロスバとスヌープバスで結合している．フロントサイドバスの帯域幅は限られているため，Tag メモリを導

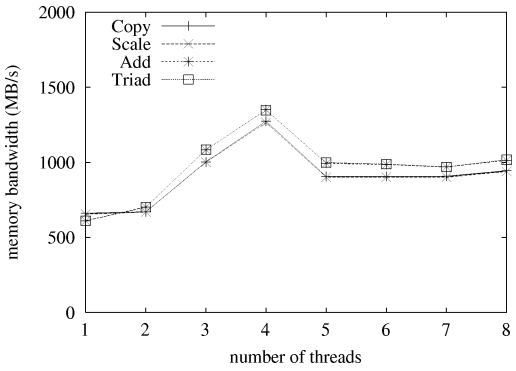

図 14 STREAM benchmark performance (MB/s) with array size 20,000,000 on NEC AzusA (with memory affinity).

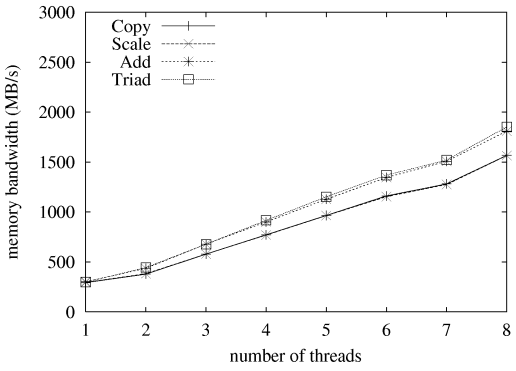

図 15 STREAM benchmark performance (MB/s) with array size 80,000,000 on SGI Origin 2000.

入して DMA 転送, 及びリモートセルからのリクエストに対するコヒーレンシを保証するためのスヌープを最小限に抑制している. Level 1,2 BLAS 型の演算では, シーケンシャルな読み込みがメモリに集中するため, Tag メモリのスワップが多発し, フロントサイドバスのアドレスバスのスループットが半減する可能性がある. フロントサイドバスが飽和状態になると, セル内のスケールが制限されるとともに, スヌープバス上のリクエスト競合が多発し, 全体のスケーラビリティに影響を与える. AzusA ではビジネスアプリケーション向けに対称型マルチプロセッサに近い性能を達成するため, レイテンシを優先した構成を取っており, この特性は前節の評価結果とも一致する. したがって, 本研究のようにメモリトラフィックの大きなルーチンを実行する場合には, 上記のように性能に問題が生じる場合があると考えられる.

ただし, 起動直後の AzusA 上においては, 図 16 に示すように高い性能を示すことが分かっており, 起動後に生成されるバッファキャッシュが性能に影響を与える可能性も考慮する必要があると思われる. この点は今後の課題である.

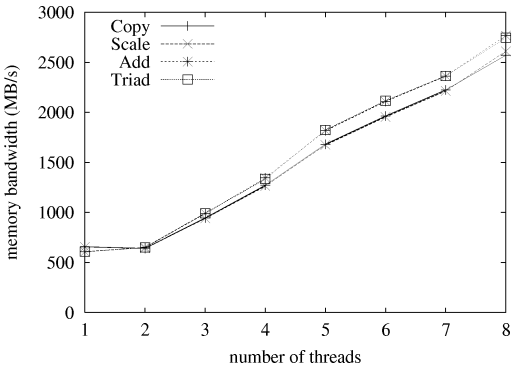

図 16 STREAM benchmark initial performance (MB/s) with array size 20,000,000 on NEC AzusA (with memory affinity).

5. ま と め

本稿では, コモディティハードウェアによる分散共有メモリ型アーキテクチャ構築のための予備評価として, NEC の Itanium ccNUMA サーバ AzusA を用いて, 疎行列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を行った.

疎行列アルゴリズムにおいて必要となる低レベルな BLAS 演算の並列化を考える場合, レイテンシを小さくするとともに, セル間でのメモリ帯域幅を十分に確保することが必要となる. 実験から, コモディティ分散共有メモリの構築に際しては, メモリ帯域幅の確保に注意する必要のあることが分かったが, オペレーティングシステムについてもより効率的な実装手法を明らかにしていく必要がある. プロセッサレベルでの最適化☆ と合わせ, 今後より厳密な評価を行っていきたい.

謝辞 本研究を進めるに当たり, 日本電気株式会社より様々なサポート及びアドバイスを頂きました. また実験の一部には産業技術総合研究所の SGI Origin 2000 を使用させて頂きました. 感謝致します. なお, 本研究の一部は, 科学研究費補助金基盤研究 (B) 13480080 及び特定領域研究 (2) 14019030 によるものである.

参 考 文 献

1) *Atlas Project Site*, http://atlas-64.sourceforge.net/.
2) *STREAM: Sustainable Memory Bandwidth in High Performance Computers*, http://www.cs.virginia.edu/stream/.
3) F. Aono and M. Kimura, *The AzusA 16-*

☆ ただし Level 1, 2 BLAS において, ATLAS による最適化を同時に行う場合, 上記の並列化手法を前提とすると, 静的にパラメータを決定することができない. 一方, ブロック単位での並列化は動的な負荷分散が容易であるが, オーバヘッドが大きくなる可能性があるため, 注意が必要である.

Way Itanium Server, IEEE Micro, 20 (2000), pp. 54–60.

4) L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, 1997.

5) J. M. Bull, *Measuring Synchronisation and Scheduling Overheads in OpenMP*, in Proceedings of the First European Workshop on OpenMP, 1999, pp. 99–105.

6) D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: a hardware/ software approach*, Morgan Kaufmann, 1999.

7) E. Focht, *Node affine NUMA scheduler*, http://home.arcor.de/efocht/sched/.

8) J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, *The Stanford FLASH multiprocessor*, in Proceedings of the 21th Annual International Symposium on Computer Architecture, 1994, pp. 302–313.

9) L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN usage*, ACM Trans. Math. Soft., 5, pp. 308–323.

10) D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, *The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor*, in Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990, pp. 148–159.

11) G. F. Pfister, *In Search of Clusters*, Prentice-Hall, second ed., 1998.

12) B. C. Whaley and J. Dongarra, *Automatically Tuned Linear Algebra Software*, in Proceedings of SC98, 1998.

13) B. C. Whaley, A. Petitet, and J. Dongarra, *Automated Empirical Optimization of Software and the ATLAS Project*, Tech. Rep. UT CS-00-451, LAPACK Working Note #147, University of Tennessee, 2000.

14) 西田晃, 小柳義夫, *OpenMP* を用いた *Jacobi-Davidson* 法の並列実装とその性能評価, 2002 年並列処理シンポジウム論文集, (2002), pp. 79–86.