計算機のメモリ階層構造を考慮した実装手法

安井　雄一郎，藤澤　克樹

近年の計算機技術の発展や，数理科学分野におけるアルゴリズムの進歩により，以前では考えられない規模の問題を扱うことができるようになってきた．その一方で，実装したソフトウェアが期待される性能を示さないといった場面も少なくない．本稿ではなぜそのような状況になってしまうのか，現在主流となる NUMA アーキテクチャを有したプロセッサの特性を示し，高速に動作することが求められるアルゴリズム実装の際にどのような点を考慮しながら進めれば良いか，それらの改善方法について解説を行う．

キーワード：計算機のメモリ階層構造，高性能計算，グラフ処理

1. はじめに

近年の計算機技術の発展はめざましく，「集積回路上のトランジスタ数は 18 カ月ごとに倍になる」というムーアの法則 (Moore's law) に追従するようにピーク性能は年々向上している．プロセッサの開発計画では，消費電力の増大や排熱の問題を伴う動作周波数の引き上げではなく，1 Hz あたりの処理性能を向上する，また CPU ソケットに搭載されているコア数を増大する，という方針で進められている．そのため現在の計算機上で高速に動作するソフトウェアを開発するためには複数コアを使用した並列計算が必須となるが，単純な並列計算では期待される性能改善に至らない状況がしばしば報告されている．本稿では大規模なデータ空間に対して計算量の小さい演算を多数繰り返すようなアルゴリズムを実装する際にどのような点に気をつければよいかをまとめる．なお本稿で扱う内容は同様の特性を持つほかのアプリケーションに対して適用可能であり，一般性を失わないように配慮した．

2. 計算機のメモリ階層構造

計算機は，図 1 のような階層構造で表現することができる．このモデルは非常に単純であるが計算機の特徴をよく捉えている．論理演算や四則演算などの演算はすべて，最上位に配置しているレジスタ (Register) 上で行われるため，メインメモリ (RAM) に配置しているデータをレジスタまでに送り届ける必要がある．その際，レジスタとメインメモリ間のデータ転送速度の差を吸収するため，Level 1 (L1)，L2（場合によっては L3，L4 も）となるキャッシュメモリが配置されている．本稿ではメインメモリより上位の階層に焦点を絞る．また，TLB (translation look-aside buffer) は，オペレーティング・システム (OS) が管理している論理アドレスと，データアクセス時に必要な物理アドレスを変換する際に用いるテーブルである．

一般的にキャッシュメモリや TLB を直接制御する方法は用意されていないが，以下の特性について考慮することで計算機の性能を引き出すことが可能になる．

- 上位の階層になるほど転送速度は高速だが記憶容量が小さく，下位の階層になるほど転送速度は低速だが記憶容量が大きい
- キャッシュメモリや TLB は短い間隔（時間的局所性が高い）で限定された範囲（空間的局所性が高い）へ何度もデータアクセスするような演算に対して有効に（効率的に）動作する

3. NUMA アーキテクチャ

近年，計算機のメモリ階層構造は複雑化しており，NUMA (Non-uniform memory access) アーキテクチャが主流となっている．もちろんこのアーキテクチャでも前節のメモリ階層構造の基本的な性質はそのまま

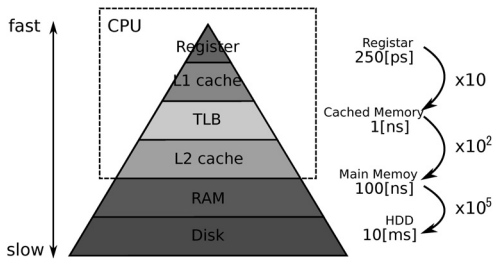
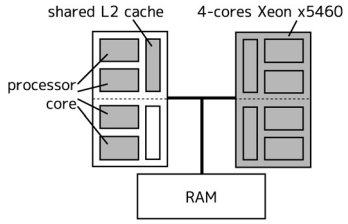


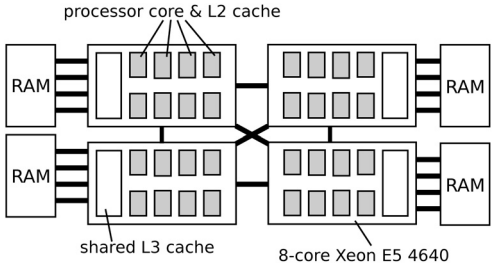
図 1　計算機のメモリ階層構造

やすい　ゆういちろう，ふじさわ　かつき
九州大学マス・フォア・インダストリ研究所
〒 819–0395 福岡県福岡市西区元岡 744



プロセッサ名	Intel Xeon X5460
アーキテクチャ	Harpertown
CPU ソケット数	2
CPU ソケットあたりの物理コア数	4
物理コアあたりのスレッド数	1
論理コアの総数	8 (= 2 × 4 × 1)

図 2　2-way Intel Xeon X5460



プロセッサ名	Intel Xeon E5-4640
アーキテクチャ	SandyBridge-EP
CPU ソケット数	4
CPU ソケットあたりの物理コア数	8
物理コアあたりのスレッド数	2
論理コアの総数	64 (= 4 × 8 × 2)

図 3　4-way Intel Xeon E5 4640

引き継いでいる．NUMA アーキテクチャと区別するため，それまでのプロセッサアーキテクチャを UMA (Uniform memory access) と呼ぶことにする．説明には図 2 に示す UMA アーキテクチャと，図 3 に示す NUMA アーキテクチャの対比を用いる．

UMA アーキテクチャでは，各プロセッサコアからメインメモリまでの距離が等しい構成である．図 2 のように CPU ソケット Intel Xeon X5460 を 2 個搭載したシステムの場合，一方の CPU ソケットは他方の CPU ソケットとメモリバスを共有してメインメモリ (RAM) に接続するが，いずれのプロセッサコアと RAM の距離は一定である．

一方，NUMA アーキテクチャでは，各プロセッサコアからメインメモリまでの距離が異なる構成であり，各ソケットはそれぞれローカルメモリと接続し，NUMA ノードと呼ばれる対を構成する．一般的に NUMA ノードは CPU ソケットと同数となる．NUMA アーキテクチャを有した計算機上ではローカルなメモリへのアクセスは高速に行われるが，ほかのソケットと対となるメモリ（リモートメモリ）へのアクセスは，ローカルメモリへのアクセスに比べて低速となる．図 3 のように CPU ソケット Intel Xeon E5-4640 が 4 個搭載したシステムの場合，NUMA ノード数も 4 となる．

4. 計算機のデータ移動性能

STREAM と呼ばれるベクトル演算を使用したメモリ帯域幅ベンチマークソフトウェア [1] を用いて，データ移動性能の測定を行い，計算機の性質を明らかにしていく．STREAM ベンチマークには 4 種類の演算が含まれるが，その 1 つである Triad 演算は，大きさが n の実数ベクトル $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \boldsymbol{R}^n$ と実数定数 r を用いて，積和計算 $\boldsymbol{a} \leftarrow \boldsymbol{b} + r\boldsymbol{c}$ を行い，その際のデータ移動量と実行に要した時間から，メモリ帯域幅（1 秒間あたりのデータ移動量 bytes）を算出する．図 4 は OpenMP を用いてスレッド並列化した Triad 演算の C/C++ 言語での実装となる．このように依存関係のない単純なループであれば，実装者は小さい実装コストでスレッド並列化に対応することができる．

```
#pragma omp parallel for
  for (long i = 0; i < n; ++i) {
    A[i] = B[i] + scalar * C[i];
  }
```

図 4　OpenMP を用いたスレッド並列 Triad 演算

図 5 に 4-way Intel Xeon E5 4640 システム上での要素数 n を $n = \{2^{10}, \ldots, 2^{30}\}$ と変化させた際のスレッド並列 Triad 演算を用いてメモリ帯域幅 (GB/s) をまとめる．まず図から要素数が 2^{20} 程度の場合，高いメモリ帯域幅であると読み取ることができるが，これは要素数が小さいためにキャッシュメモリ上に格納されているためである．STREAM ベンチマークでは何度（デフォルトでは 20 回）も同じ演算を繰り返し，最小値をメモリ帯域幅として出力するためこのような現象が起きたと考えられる．また，並列計算のオーバーヘッドにより実行が安定せず傾向を読み取ることが容易ではない．一方，十分に大きな要素数に対しては，スレッド数が 16, 32, 64 のときそれぞれ，95, 98, 92 GB/s となる．しかしながら，64 スレッド並列計算時

[1] STREAM: Sustainable Memory Bandwidth in High Performance Computers　http://www.cs.virginia.edu/stream/

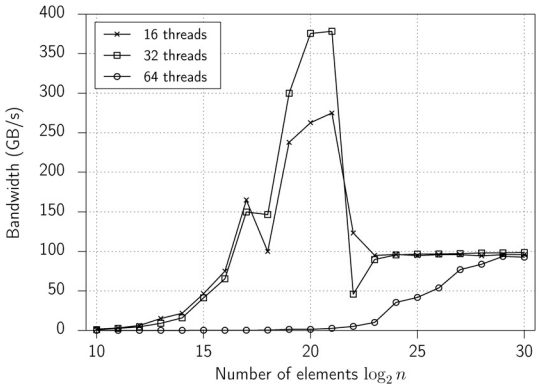


図 5 STREAM TRIAD 演算によるメモリ帯域幅

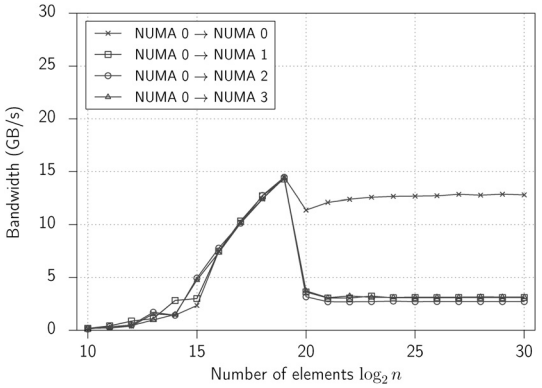


図 6 NUMA ノード 0 から各 NUMA ノードまでのメモリ帯域幅 (GB/s)

には，各コアに 2 つずつスレッド（Hyper-threading 機構）が立ち上がっている状況であるので，実コア数以上の生成されたスレッドの計算機資源の要求の衝突により，32 コアの場合と比べて効率が悪化している．

4.1 ローカルメモリとリモートメモリのデータ移動性能

詳細な性質を調べるために Linux 上で提供されている `numactl` コマンドを用いて，使用するプロセッサコアやメモリ確保を行う対象の NUMA ノードを指定して，データ移動性能を計測する．`numactl` コマンドを用いて，NUMA node 0 に配置されている 16 論理コアに 16 スレッドを固定し NUMA node 3 のローカルメモリ上へ確保したデータ領域に対するメモリ帯域幅測定を行うためには，次のように指定すれば良い．ここで，`--physcpubind --membind` の指定に用いるのは NUMA ノードの ID であるが，NUMA ノード ID は環境によって異なるため注意が必要である．Linux 上の `/proc/cpuinfo` で確認することができる．記述されている `processor` の項目が論理コア ID を，`physical id` の項目が NUMA ノード ID をそれぞれ表している．また Portable Hardware Locality (HWLOC) ライブラリ [1] を用いると良い．

```
OMP_NUM_THREADS=16 numactl --physcpubind=0 \
                           --membind=3 ./stream
```

図 6 は要素数を $n = \{2^{10}, 2^{11}, \ldots, 2^{30}\}$ とした際の指定した NUMA ノード間のメモリ帯域幅をまとめたもので，各 NUMA ノードに確保した領域に対し，NUMA ノード 0 に配置している 16 論理コアに固定した 16 スレッドを用いて Triad 演算を行った際のメモリ帯域幅を測定した．図から同一の NUMA ノード内での通信では 12 GB/s を超えるメモリ帯域幅となるが，NUMA ノードを横断した通信では 3 GB/s と NUMA 内に比べて約 1/4 の性能となってしまうことが確認できる．

4.2 メモリ確保方針：ローカル確保

ローカル確保とはメモリ確保する際に，使用しているスレッドが配置されているコアに近いメモリ（ローカルメモリ）から割り当てを行うメモリ確保方針である．基本的なメモリ確保方針であり何も設定しなければこの設定が有効になっている場合が多い．`numactl` コマンドでは `--localalloc` オプションで指定できる．32 スレッドを NUMA ノード 0, 1 の論理コア 32 コアに固定し，メモリ確保先をそれぞれのローカルメモリとする指定には以下のようにすれば良い．

```
OMP_NUM_THREADS=32 numactl --physcpubind=0,1 \
                           --localalloc ./stream
```

4.3 メモリ確保方針：インターリービング

広域なメモリ空間にデータアクセスする際に，アクセスコストが異なるローカルアクセスとリモートアクセスが混在すると負荷分散に偏りが生じてしまう．インターリービングでは，広域なデータ領域を確保する際に，領域をページ単位（通常は 4 KBytes）で指定した NUMA ノードに配置したメモリへラウンドロビン方式で分散する．このメモリ確保方針を用いることで，各コアの各 NUMA ノードへのデータアクセス量を平準化して，前述の負荷分散の偏りを緩和することができる．`numactl` コマンドでは `--interleave` オプションで指定できる．32 スレッドを NUMA ノード 0, 1 の論理コア 32 コアに固定し，インターリービングの

対象を NUMA ノード 0, 1 とする指定には以下のようにすれば良い．

```
OMP_NUM_THREADS=32 numactl --physcpubind=0,1 \
                   --interleave=0,1 ./stream
```

4.4 メモリ確保方針による帯域幅の比較

図 7(a) と 7(b) に STREAM の TRIAD 演算を用いたメモリ確保方針ごとのメモリ帯域幅 (GB/s) をまとめる．いずれも NUMA ノード数を 1, 2, 4 と，要素数を $n = \{2^{10}, \ldots, 2^{30}\}$ と変化させている．ここでスレッド数は NUMA ノードに配置している論理コア数と同数に指定している．この実験においても，ローカル確保 (Local-allocation)，インターリービング (Interleaving) いずれにおいても要素数が 2^{20} 以下ではスレッド並列のオーバーヘッドやキャッシュメモリの効果により，現象を理解することが困難であることを述べておく．NUMA ノード数を 1（16 スレッド），2（32 スレッド），4（64 スレッド）と増加させた際のメモリ帯域幅は，Local allocation では，13 GB/s, 21 GB/s, 24 GB/s と規則的に増大するが，Interleaving では，13 GB/s, 6 GB/s, 8 GB/s と変化する．このように Interleaving によるメモリ確保では帯域幅を平準化しているため，単純に増大するわけではない．

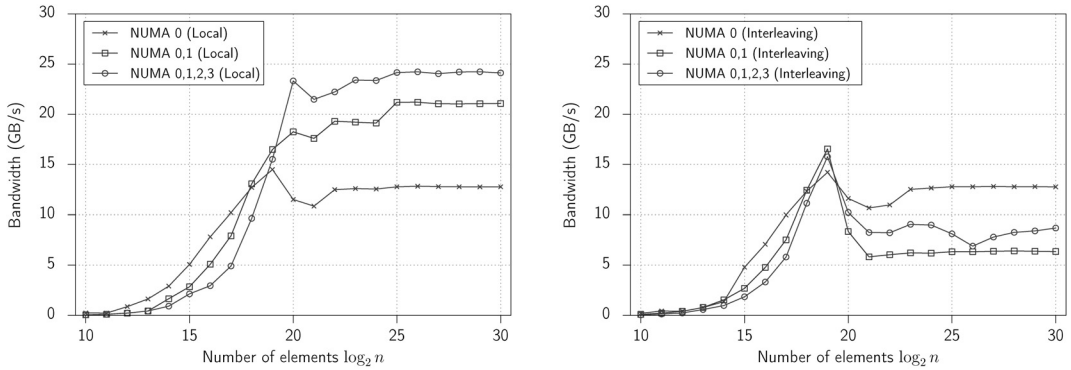
本来 TRIAD 演算（正確には STREAM の 4 種類の演算すべて）では，時間的局所性，空間的局所性が高いデータ・アクセスを行っているため Local allocation を選択すれば良い．反対に，広域にわたりデータアクセスが必要なアルゴリズム実装に対しては Interleaving を選択することが望ましい．図 6 で計測したようにこのシステム上ではローカルメモリへのアクセスとリモートメモリへのアクセスの性能差は約 4 倍にもなるため，Local allocation では負荷分散の偏りにより最も遅い箇所に性能が律速され，性能を引き出すのは難しいと予測される．

5. NUMA を考慮した高速化

前節では NUMA アーキテクチャの特徴をメモリ帯域幅から示した．本節では，アルゴリズムやデータ構造を考慮して，コストの大きなリモートメモリへのデータアクセスを削減による性能向上した事例を紹介する．前節ではソースコードの編集を行わずに `numactl` を用いたスレッドの論理コアへの固定やメモリ確保の指定を行い，特性や性質を観察した．より細かい制御をするためには，通常の Linux で用意された `sched_setaffinity()`，`sched_getaffinity()`，`mbind()` といった関数群を用いる必要がある．`sched_setaffinity()` は（関数を発行した）現在のスレッドがどの論理コア上に固定されているか確認するための関数で，`sched_setaffinity()` は現在のスレッドを論理コア上に固定する関数である．また，`mbind()` は指定したメモリ領域を指定した NUMA ノード上に固定する関数である．なお，これらの関数を用いるために必要なソースコードの修正に関する解説や具体的な例については本節では省略し，主に NUMA を考慮するためにどのようにアルゴリズムやデータ構造を修正を行ったかに留めることにする．

5.1 STREAM の TRIAD 演算

前節まで扱った TRIAD 演算では入力となる実数ベクトル $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ はそれぞれ 1 本の配列で実装している．本節では NUMA ノードごとに担当する範囲に分割して，各部分配列をローカルメモリ上に確保するように修正を行った．このように修正を行うと，データアク



(a) Local-allocation　　(b) Interleaving

図 7　STREAM TRIAD: メモリ確保方針ごとのメモリ帯域幅 (GB/s)

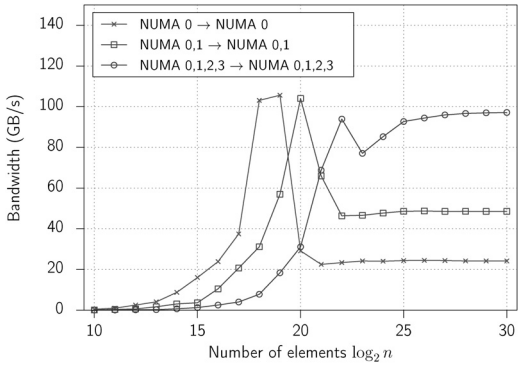


図 8　修正版 TRIAD 演算によるメモリ帯域幅測定

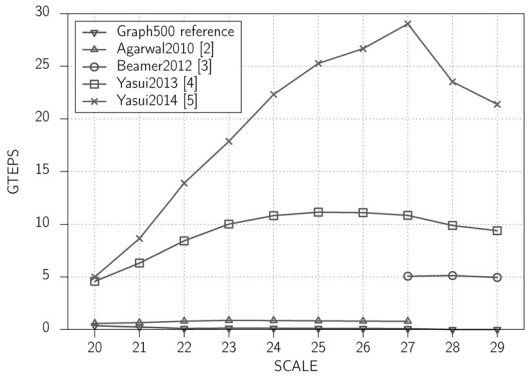


図 9　先行研究における幅優先探索性能

セスを NUMA ノード単位で完全に独立させることができる．図 8 は修正版 TRIAD 演算による結果である．NUMA ノード数を 1, 2, 4 と増加させると，それに従い 24, 48, 96 GB/s とほぼ線形にメモリ帯域幅が増大している．また，図 5 で示した結果と比べて，メモリ帯域幅が安定しており性能予測も容易になる．

5.2　幅優先探索

近年，グラフを用いた解析手法はさまざまな分野で盛んに研究されており，特に幅優先探索 (Breath-first search; BFS) は基本的かつ重要なグラフ処理である．BFS は入力グラフ $G=(V,E)$ の点数 $n=|V|$ と枝数 $m=|E|$ を用いて線形の計算量 $O(n+m)$ となるシンプルな処理であるが，多くのグラフアルゴリズムのサブルーチンとして頻繁に用いられること，グラフ処理の重要性が高まっているという状況から，高速化に対する関心は非常に高い．

近年，HPC 分野においても Graph500 と呼ばれるグラフ性能を用いたベンチマークが提案されており，1 台計算機の内部でのスレッド並列計算だけでなく，スーパーコンピュータ上での高速なグラフ探索に関して盛んに議論されている．Graph500 ベンチマーク[2] は BFS の探索性能指標を用いて計算機の不連続なメモリアクセス性能の評価を目的として設計が行われ，2010 年 11 月から開始され半年ごとに更新される．このベンチマークでは 2 種類のパラメータ SCALE と edgefactor（$=m/n$，デフォルトでは 16）を入力として手順 (a), (b), (c) から構成される．(a) 点数 $n=2^{\mathrm{SCALE}}$，枝数 $m=n\cdot$edgefactor となる無向グラフ Kronecker graph の枝リストを生成する．(b) 生成した枝リストからグラフ表現を構築する．(c) 次の手順を 64 回繰り返す；構築したグラフ表現に対する BFS を行い，1 秒あたりの探索枝数 (traversed edges per second; TEPS) を算出する．リストの順位は (c) で得られる 64 個の TEPS 値の中央値により決定される．また Green Graph500 ベンチマーク[3] では，消費電力あたりの Graph500 における TEPS 値となる電力性能指標 TEPS/W によりリストの順位を決定する．

図 9，表 1 で示すように，BFS に対する並列アルゴリズムは，始点から距離 (Level) ごとに，未探索点を並列に探索していく Level-synchronized BFS を基本に提案・改善されている．Beamer らのアルゴリズム [3] は，探索済領域の前線となる点集合から未探索点を探索する Top-down アルゴリズムと，未探索点から探索済領域の前線を探索する Bottom-up アルゴリズムを組み合わせて，直径の小さい Small-world 性を有するグラフに対して高い性能を示している．探索済領域の前線が小さい場合には Top-down アルゴリズムを選択し，そうでない場合には Bottom-up アルゴリズムを選択することで，枝探索を入力グラフの枝数の数% 程に削減することができる．Beamer らは点数 2^{28}，枝数 2^{32} からなる Kronecker graph に対し 4-way Intel Xeon E7-8870 を用いて 5.1 GTEPS (10^9 TEPS) を達成している．著者らはこのアルゴリズムに対して NUMA を考慮した高速化を行い，同等の計算機環境上で 2.2 倍となる 11.15 GTEPS を達成した [4]．さらに Bottom-up アルゴリズムのデータアクセスパターンと，Small-world グラフの形状を考慮したグラフ表現方法を適用して 2.68 倍の改善を提案している [5]．

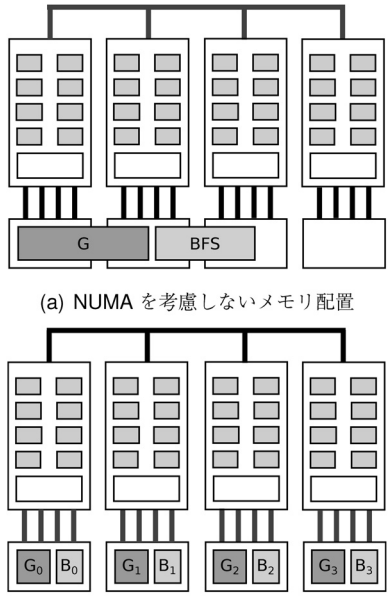
それでは，我々が文献 [4, 5] で用いたグラフ表現方法の概要について説明を行う．我々は基本的に CSR (Compressed Sparse Row) 形式と呼ばれる隣接リス

[2] Graph500: http://www.graph500.org.

[3] Green Graph500: http://green.graph500.org.

表 1　先行研究における幅優先探索性能

実装者	計算機環境	(n, m)	TEPS
Madduri	Cray MTA-2 (40 procs)	$(2^{21}, 2^{30})$	0.5 G
Agarwal [2]	Intel Xeon X7560×4	$(2^{20}, 2^{26})$	1.3 G
Beamer [3]	Intel Xeon E7-8870×4	$(2^{28}, 2^{32})$	5.1 G
Yasui [4]	Intel Xeon E5-4640×4	$(2^{26}, 2^{30})$	11.1 G
Yasui [5]	Intel Xeon E5-4640×4	$(2^{27}, 2^{31})$	29.0 G



(a) NUMA を考慮しないメモリ配置

(b) NUMA を考慮したメモリ配置

図 10　NUMA アーキテクチャと BFS に用いるグラフや作業のデータ配置

ト表現を用いているが，このグラフ表現では隣接点はすべて連続に並びデータ・アクセスの効率が良い．Graph500 ベンチマークでは各ローカルメモリに収まらない巨大なグラフを用いるため，図 10(a) のように複数の NUMA ノードに横断してデータ領域を確保してしまう．一方で，図 10(b) のように各ローカルメモリに分割し，データアクセスをローカルメモリ上へのみに制御することで高い性能を引き出すことができる．我々は入力グラフを以下に示す分割手法を適用することで，各点から未探索の隣接点をたどる処理をローカルメモリ上へのメモリアクセスのみで行うように制御した．

NUMA ノード数が ℓ となるシステム上に，入力グラフ G の ℓ 個の部分グラフ $\{G_k\}, (k = \{0, 1, \ldots, \ell-1\})$ への分割を考える．ここで NUMA ノード k に部分点集合 V_k と隣接リストの部分集合 A_k に配置するものとして，部分点集合 V_k を以下のように定義する．

$$V_k = \left\{ v_j \in V \mid j \in \left[\frac{kn}{\ell}, \frac{(k+1)n}{\ell} \right) \right\}$$

さらに隣接リスト A は，Top-down 探索で使用する各点 $v \in V$ から出ていく隣接枝をまとめた $A_k^{\mathcal{F}}(v)$ と，Bottom-up 探索で使用する各点 $w \in V_k$ に入ってくる隣接枝をまとめた $A_k^{\mathcal{B}}(w)$ に，それぞれ分割する．これらは分割数 ℓ が 1 であるとき一致するが，それ以外では入力グラフが無向グラフであったとしても，$A_k^{\mathcal{F}}(v)$ と $A_k^{\mathcal{B}}(w)$ は一致しない．

$$A_k^{\mathcal{F}}(v) = \{ w \mid w \in \{V_k \cap A(v)\} \},\ v \in V,$$
$$A_k^{\mathcal{B}}(w) = \{ v \mid v \in A(w) \},\ w \in V_k.$$

以上の分割手法を用いることで NUMA を考慮した効率的な探索が可能となる．

最新の Graph500 リスト（2014 年 6 月発表）[4] ではさまざまな実行環境で大幅に性能向上している．特に，SGI UV2000 という共有メモリ型のスーパーコンピュータを用いて，点数 2^{32}，枝数 2^{36} の Kronecker グラフに対して，世界最大規模の 640 スレッド並列計算を用いて共有メモリ型計算機で最高性能となる 131.4 GTEPS を記録した．また，Green Graph500 リスト（2014 年 6 月発表）[5] の Big Data category では，4-way Intel Xeon E5-4640 サーバを用いて点数 2^{30}, 枝数 2^{34} のグラフに対して 28.5 GTEPS と 59.1 MTEPS/W を達成し世界 1 位を獲得した．なお，前述の UV 2000 の結果は商用スパコンでは最高の電力性能と示されている．

5.3　その他の高速化事例

幅優先探索以外にも，我々は高性能な汎用半正定値問題ソルバ SDPARA (SemiDefinite Programming Algorithm PARAllel version) [6] や SDPA (Semidefinite Programming Algorithms)，データ構造に ZDD (Zero-suppressed decision diagram) を用いた格子グラフに対するパスの数え上げ問題 [7]，最短路計算や中心性計算 [8] などに対して，NUMA を考慮した高速化を適用して性能改善に成功している．詳しくは各文献を参考にしていただきたい．なお，我々は研究で得られた知見を ULIBC (Ubiquity Library for Intelligently Binding Cores) とライブラリ化しており，準備が整い次第公開する予定である．

[4] http://www.graph500.org/results_jun_2014
[5] http://green.graph500.org/list_2014_06_isc.php

6. おわりに

本稿では計算機のメモリ階層構造が，計算機実験において性能上に与える影響について，実際に計算機実験を用いて性質を明らかにした．特に NUMA アーキテクチャでは計算機の内部にアクセスコストが小さいローカルメモリと，アクセスコストが大きいリモートメモリが存在することを説明し，メモリ帯域幅を測定することでそれらの性質を明らかにした．さらに，現在我々が課題としているグラフ処理に対する NUMA を考慮した高速化について解説を行った．本稿で扱った内容は現在主流のプロセッサ・アーキテクチャに対するものだが，取り上げた手法は汎用性を失わずに広く適用が可能である．今後も階層の複雑化が進むこと予想されており，本稿のような実装手法の重要性はより高まっていくと予想されている．読者の皆様の参考になれば幸いである．

謝辞　本稿を執筆する機会をいただきました国立情報学研究所の宇野毅明先生をはじめ，編集委員の先生方にこの場を借りて御礼申し上げます．なお，本研究は科学技術振興機構 (JST) の戦略的創造研究推進事業「CREST」における「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」によるものである．また支援いただいた統計数理研究所，日本 SGI 株式会社，Silicon Graphics International Corp. に御礼申し上げます．

参考文献

[1] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault and R. Namyst, “hwloc: A generic framework for managing hardware affinities in HPC applications,” *Proc. IEEE Int. Conf. PDP2010*, 2010.

[2] V. Agarwal, F. Petrini, D. Pasetto and D. A. Bader, “Scalable graph exploration on multicore processors,” *Proc. ACM/IEEE Int. Conf. SC10*, 2010.

[3] S. Beamer, K. Asanović and D. A. Patterson, “Direction-optimizing breadth-first search,” *Proc. ACM/IEEE Int. Conf. SC12*, 2012.

[4] Y. Yasui, K. Fujisawa and K. Goto, “NUMA-optimized parallel breadth-first search on multicore single-node system,” *Proc. IEEE Int. Conf. BigData 2013*, 2013.

[5] Y. Yasui, K. Fujisawa and Y. Sato, “Fast and energy-efficient breadth-first search on a single NUMA system,” *Proc. IEEE Int. Conf. ISC’14*, 2014.

[6] K. Fujisawa, T. Endo, Y. Yasui, H. Sato, N. Matsuzawa, S. Matsuoka and H. Waki, “Peta-scale general solver for semidefinite programming problems with over two million constraints,” *Proc. IEEE Int. Conf. IPDPS 2014*, 2014.

[7] 安井雄一郎，藤澤克樹，竹内聖悟，湊真一，“ULIBC ライブラリを用いた共有メモリ型並列アルゴリズムの高速化，”2014 年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.

[8] Y. Yasui, K. Fujisawa, K. Goto, N. Kamiyama and M. Takamatsu, “NETAL: High-performance implementation of network analysis library considering computer memory hierarchy,” *J. Oper. Res. Soc. Jpn.*, **54**, 259–280, 2011.