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

安井 雄一郎[1,2,a)]　藤澤 克樹[1,2,b)]　竹内 聖悟[3,4,c)]　湊 真一[3,4,d)]

概要：現在, 共有メモリ型並列計算機の主要なアーキテクチャとなっている NUMA (Non-Uniform Memory Access) アーキテクチャを有する計算機上には, 各プロセッサと対となるローカルメモリと, 他のプロセッサと対となるリモートメモリが存在し, 各プロセッサコアと各メインメモリ間の距離が異なるためアクセスコストが均一でない. NUMA アーキテクチャ上での高速化にはコストの大きなリモートメモリへのアクセスを削減することが重要になるものの, 並列実行時の各スレッドが計算機トポロジ上のどこに配置しているかといった NUMA アーキテクチャを考慮した制御に必要な情報の取得は, 既存ライブラリ群だけでは容易ではない. 本研究では CPU アフィニティやローカルメモリ確保などの機能をまとめたライブラリ ULIBC (Ubiquity Library for Intelligently Binding Cores) を開発し, いくつかの並列アルゴリズムに対して, ULIBC を用いた NUMA アーキテクチャを考慮した高速化を示す.

キーワード：スレッド並列計算, CPU アフィニティ, グラフ処理, ZDD, 数理最適化問題

Fast implementation of shared-memory parallel algorithm using ULIBC (Ubiquity Library for Intelligently Binding Cores)

Yuichiro Yasui[1,2,a)]　Katsuki Fujisawa[1,2,b)]　Shogo Takeuchi[3,4,c)]　Shin-ichi Minato[3,4,d)]

Abstract: Under NUMA (Non-uniform memory access) which is recent major shared memory multi-core architecture, each processor has its own local memory faster than non-local memory. Some speedup techniques which based on reducing access costs to the non-local memory larger than local memory considering NUMA architecture require development cost due to not enough to functions of some libraries for managing of memory access. Therefore, we implemented ULIBC (Ubiquity Library for Intelligently Binding Cores) for CPU affinity and local memory allocation. Finally, we show that some applications achieved performance improvements using ULIBC.

Keywords: thread parallel computing, CPU affinity, graph processing, ZDD, mathematical optimization

1. はじめに

現在, 共有メモリ型並列計算機の主要な NUMA (Non-Uniform Memory Access) アーキテクチャを有する計算機上には, プロセッサと対となるローカルなメインメモリ (ローカルメモリ) と, 他のプロセッサと対となるメインメモリ (リモートメモリ) が存在する. プロセッサとローカルメモリの対は NUMA ノードと呼ばれ, 一般的に NUMA ノード内のローカルメモリへのアクセスコストは, NUMA ノードを横断した他のプロセッサと対となるリモートメモリへのアクセスコストよりも小さい. このように以前までの UMA (Uniform Memory Access) アーキテクチャを有する計算機での均一なメモリアクセスと比べて, NUMA アーキテクチャを有する計算機でのメモリアクセスは複雑かつ不均一化している. そのため, 並列数の増加に従う並列効

[1] 中央大学
Chuo University
[2] JST CREST
[3] 北海道大学
Hokkaido University
[4] JST ERATO
[a)] yasui@indsys.chuo-u.ac.jp
[b)] fujisawa@indsys.chuo-u.ac.jp
[c)] takeuchi@erato.ist.hokudai.ac.jp
[d)] minato@ist.hokudai.ac.jp

率の低下を抑えることは難しく, 特に演算量に比べてデータ移動量が大きい並列アルゴリズムではその傾向が強い.

NUMA アーキテクチャを考慮した高速化に関する先行研究 [1], [5], [6], [7], [9] では, コストの大きなリモートメモリへのメモリアクセスを削減し, ローカルメモリへのメモリアクセスの局所性を高めて, 性能改善を達成している. それらの制御には CPU アフィニティ設定やメモリ領域のローカルメモリへの固定などが必要だが, Portable Hardware Locality (hwloc) [2], Likwid [3], Intel コンパイラの Thread Affinity Interface [4], OpenUH コンパイラ [5], Linux 上での `numactl` コマンド, `libnuma` ライブラリ, `sched_{get,set}affinity` や `mbind` などのシステムコール関数などが既に存在する. しかしながら, これらの環境だけでは著者らが文献 [9] で提案したアルゴリズムの実装に必要な「スレッド並列での実行中の各スレッドが何番目の NUMA ノードの何番目のコア上に固定されるか」といった情報の取得は容易に実現することができない.

そこで本研究では, NUMA アーキテクチャ計算機上で汎用的な高速化を行うための, アフィニティ設定やローカルメモリ設定などの機能をまとめたライブラリ ULIBC (Ubiquity Library for Intelligently Binding Cores) を開発した. ULIBC は計算機上の CPU ソケット, 物理コア, 論理コアなどの CPU トポロジ情報を取得し, トポロジ情報からスレッドと論理コアの割当表を作成する. その後, 割当表を参照して, ローカルメモリへのメモリ確保や, スレッド並列時に各スレッドを論理コア上への固定を行う. ULIBC は割当表の参照によるシンプルな制御を行っており, 以下のような利点が挙げられる.

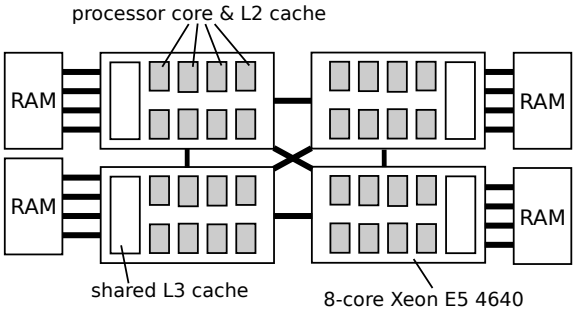
- 各スレッドの論理コアへの割当は事前に決定するため, スレッド並列時における各スレッドが固定される位置の把握や性能予測が容易である
- 並列時のアフィニティ設定のためのオーバーヘッドはシステムコールの発行コスト以外ほぼ発生しない
- 割当表を参照するための関数を挿入すれば良いため, 実装上の手間を抑えられる

加えて本論文では, いくつかの性質の異なるいくつかの並列アルゴリズムに対し, ULIBC を用いた NUMA アーキテクチャを考慮した高速化を示す. 対象は以下の 4 種類でいずれも高速化を達成し, 計算機上のボトルネック箇所の特定を行った.

- メモリバンド幅に対する STREAM ベンチマーク
- Graph500 ベンチマークの対象であるグラフに対する並列幅優先探索 (BFS) [9]
- グラフ上の閉路を持たないパスを列挙する並列アルゴリズム Edge-based approach (EBA) [13]
- 近年, 最も注目されている数理最適化問題の一つである半正定値計画問題に対する高性能汎用ソルバ SDPA [14]

2. NUMA アーキテクチャ上でのメモリアクセス

NUMA アーキテクチャにおける並列アルゴリズムのメモリアクセスに関して説明を行う. まず図 1 は NUMA アーキテクチャである Intel Xeon E5-4640 が 4 基搭載された共有メモリ型計算機環境 4-way Sandybridge-EP である. この計算機環境において, まず 4 基ある CPU ソケットはそれぞれ 8 個の物理コアを持ち, 各物理コアは個別の L1/L2 キャッシュを持ち, 同時に 2 つのスレッド (論理コア) を実行することが可能である. また, ソケット上には 16 の論理コア (8 の物理コア) で共有する L3 キャッシュが配置している.



プロセッサ	Intel Xeon E5-4640 (8-core)
ソケット数	4
論理コア数	64
物理コア数	32
スレッド数/物理コア	2

図 1 計算機環境 4-way Sandybridge-EP

続いて, 4-way Sandybridge-EP 上での動的メモリ確保に関して説明する. 特殊な設定を行っていなければ (`get_mempolicy` 関数で得られるメモリ確保のポリシーが `MPOL_DEFAULT` ならば), C/C++ 言語の `malloc`/`new` 関数を用いた動的メモリ確保は, 関数呼び出しを行ったコアのローカルメモリへ連続した領域を試みる. その際, 確保する領域が 1 つのローカルメモリ上に収まりきらなければ, 図 2 のように他のメモリに対象を移し, 先程と同様に連続した領域を試みる. このように確保された領域はサイズによっては複数のローカルメモリを横断してしまい, コストが小さいローカルメモリへのアクセスと, コストが大きいリモートメモリへのアクセスが混在することになり, 負荷分散が難しく並列効率が下がる原因となる. そのため, 並列計算での各スレッド間でのメモリアクセス要求の衝突や, 各 NUMA ノード間を接続するインターコネクトに非常に大きな負荷がかかることになる. しかしながら, 実際にはメモリ領域はページ単位でファーストタッチに基づいて確保されるため, 確保されるメモリ領域がいずれの NUMA ノードに配置されるかはどのように初期化するかに依存する.

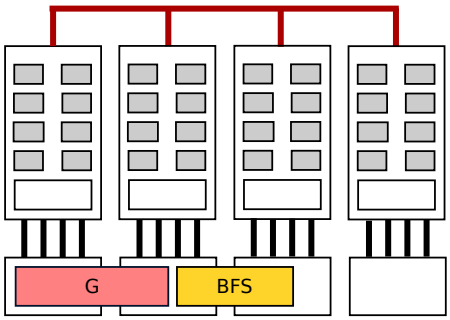


図 2 一般的なメモリ確保と複数ローカルメモリへのメモリアクセス

図 3 のようにアクセスするメモリ領域を各ローカルメモリに分割することができ, メモリアクセスの局所性が高いアルゴリズム設計を行うことができるならば, 各スレッドがアクセスコストの小さいローカルメモリへのアクセスするように制御を行うことで, 不均質なデータアクセスによる性能低下やメモリアクセス要求の衝突による性能低下を抑えることができる.

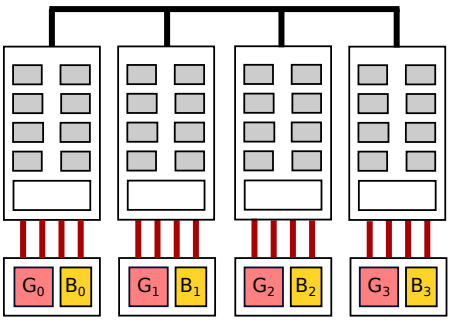


図 3 NUMA を考慮したメモリ確保とローカルなメモリアクセス

3. ULIBC ライブラリ

本研究で実装したライブラリ ULIBC は, CPU アフィニティとローカルメモリへの動的メモリ確保などの制御を容易に行うためのライブラリである. 類似のライブラリ等は既に存在するものの, 既存のライブラリだけでは「スレッド並列計算時に各スレッドを実行する論理コアが何番目の CPU ソケットの何番目の物理コア上に固定されるか」といった情報を取得することは容易ではない.

詳細なトポロジ情報の取得には Portable Hardware Locality (hwloc) [2], Likwid [3] が挙げられるが, アフィニティ設定を行った後に実際にどのコア上に固定されているかという機能は提供されていない. また, コンパイラによるアフィニティ設定として, GCC コンパイラの OpenMP ライブラリ, Intel コンパイラ Thread Affinity Interface [4], OpenUH コンパイラ [5] などが挙げられるが, GCC と ICC では環境変数によるアフィニティ設定のみで, OpenUH ではアフィニティ設定やローカルメモリへのメモリ確保の両方を提供しているものの独自プラグマ文による指定となり環境に依存してしまう. コマンドラインツールによるアフィニティ設定やローカルメモリへのメモリ確保設定として, Linux 上での `numactl` コマンドが挙がられるが, プログラムの実行者が事前に計算機トポロジ情報を把握している必要がある. さらに一般的な Linux 上で利用可能な `libnuma` ライブラリ, `sched_{get,set}affinity` や `mbind` などのシステムコール関数を用いると, 自由な制御ができるものの実装コストが大きく実装が煩雑になってしまう.

既存のライブラリに対し ULIBC は, 一般的な Linux 環境を対象とし, ULIBC を適用したアプリケーションが実行すると, まず計算機トポロジ情報 (図 4 の processor topology) の自動的取得を行い, 各スレッドの論理コアへの割当表 (図 4 の mapping table) を作成する. アフィニティ設定とローカルメモリへのメモリ確保はこの割当表を参照して実現する. なお, ULIBC の利用に管理者権限は不要である.

(1) 計算機トポロジの取得.
(2) スレッド並列時に参照するための, スレッド ID と論理プロセッサ ID の割当表の作成.
(3) (2) を参照しローカルメモリ上に固定した動的メモリ領域確保 (`mmap` 関数と `mbind` 関数による制御).
(4) スレッド並列時に, (2) を参照して各スレッドを論理コア上に固定 (`sched_setaffinity` 関数による制御) し, (1) を参照して (3) で確保したメモリ領域にアクセスするように制御.

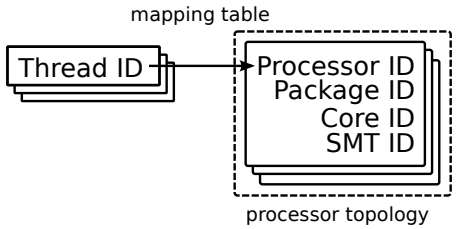


図 4 スレッド ID と論理プロセッサ ID の割当表 (mapping table) と計算機トポロジ (processor topology)

3.1 計算機トポロジの取得

ULIBC が自動的に行う計算機トポロジの取得について説明する. まず各論理コアを識別するための ID について表 1 にまとめる.

表 1 計算機トポロジを表現するための ID (括弧内の数字は 4-way Sandybridge-EP 上で取り得る範囲を表している)

ID	説明
Processor ID	論理 CPU コアに対する ID (0,1,...,63)
Package ID	CPU ソケットに対する ID (0,1,2,3) (NUMA ノードに対応)
Core ID	CPU ソケット内の物理コアに対する ID (0,1,...,7)
SMT ID	物理コア上での論理コア番号 (0,1)

計算機毎に CPU ソケット数, ソケット数内の物理コア

数, 物理コア上の論理コア数は異なり, プロセッサコア ID, CPU ソケット ID, ソケット内の ID の順序は異なる. 一般的な Linux 環境を想定すると, 汎用的な計算機トポロジの取得には以下のような方法を挙げることができる. いずれの取得方法でも同様の情報を取得することが可能で, 計算機トポロジの取得に要する実行時間はプログラムの性能に影響を及ぼさないほど小さい.

(a) デバイスファイル `/proc/cpuinfo`
Processor ID, Package ID, Core ID と, `processor`, `physical id`, `core id` がそれぞれ対応.

(b) ディレクトリ構成 `/sys/devices/system/` [9]
Processor ID, Package ID, Core ID と, `cpu`, `node` / `physical_package_id`, `core_id` がそれぞれ対応. 物理コア上に動作する Processor ID をまとめた `thread_siblings_list` が提供される.

(c) APICID のサブセット情報 [10]
各論理コアに一意の ID となる APIC ID のサブセット情報により, Processor ID, Package ID, Core ID, SMT ID を取得できる.

表 2 は, Hyper-threading 機構を有効とした論理コアが 64 となる 4-way Sandybridge-EP 上の計算機トポロジを示した Processor ID, Package ID, Core ID の対応表である. これらはアプリケーションが実行時に ULIBC が自動的に取得するものである.

表 2 4-way Sandybridge-EP 上の計算機トポロジ

	Core ID							
	0	1	2	3	4	5	6	7
Package ID	Processor ID							
0	0	1	2	3	4	5	6	7
	32	33	34	35	36	37	38	39
1	8	9	10	11	12	13	14	15
	40	41	42	43	44	45	46	47
2	16	17	18	19	20	21	22	23
	48	49	50	51	52	53	54	55
3	24	25	26	27	28	29	30	31
	56	57	58	59	60	61	62	63

3.2 割当表を用いたアフィニティ設定

ULIBC におけるアフィニティ設定について説明を行う. Linux 環境での各スレッドの論理コアへの固定には `sched_setaffinity` 関数を用いて Processor ID (論理 CPU コア ID) で指定し, 動的確保したメモリ領域を特定のローカルメモリ上に固定するための関数 `mbind` を用いて Package ID (NUMA ノード ID) で指定する. ULIBC では, スレッド並列計算時の各スレッドに与えた一意のスレッド ID と Processor ID との割当表を作成し制御を行う.

ULIBC ではスレッドと論理コアを割り付けるアフィニティ設定の方針を 3 種類 Scatter, Compact, Compact+ を用意している. Scatter では使用するスレッドを NUMA ノードに分散するように, Compact では使用するスレッドを NUMA ノードに集約するように, それぞれスレッドを各論理コアに割り当てる. また Compact+ は, Compact と似たアフィニティ設定だが, 物理コア上へのスレッドの重複 (Hyper-Threading 機構の利用) は回避する割当となる.

表 3.2 と表 3.2 に 4-way Sandybridge-EP 上のアフィニティ設定毎のスレッド ID と Processor ID の割当表, アフィニティ設定毎のスレッド数と使用する NUMA ノード数, NUMA ノード内の物理コア数, 物理コア上で同時実行するスレッド数をまとめる.

表 3 4-way Sandybridge-EP 上のアフィニティ設定毎のスレッド ID と Processor ID の割当表

(a) Scatter アフィニティ

スレッド ID	Processor ID							
00 – 15	00	08	16	24	01	09	17	25
	02	10	18	26	03	11	19	27
16 – 31	04	12	20	28	05	13	21	29
	06	14	22	30	07	15	23	31
32 – 47	32	40	48	56	33	41	49	57
	34	42	50	58	35	43	51	59
48 – 63	36	44	52	60	37	45	53	61
	38	46	54	62	39	47	55	63

(b) Compact アフィニティ

スレッド ID	Processor ID							
00 – 15	00	01	02	03	04	05	06	07
	32	33	34	35	36	37	38	39
16 – 31	08	09	10	11	12	13	14	15
	40	41	42	43	44	45	46	47
32 – 47	16	17	18	19	20	21	22	23
	48	49	50	51	52	53	54	55
48 – 63	24	25	26	27	28	29	30	31
	56	57	58	59	60	61	62	63

(c) Compact+ アフィニティ

スレッド ID	Processor ID							
00 – 15	0	1	2	3	4	5	6	7
	8	9	10	11	12	13	14	15
16 – 31	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31
32 – 47	32	33	34	35	36	37	38	39
	40	41	42	43	44	45	46	47
48 – 63	48	49	50	51	52	53	54	55
	56	57	58	59	60	61	62	63

3.3 ULIBC を用いてアフィニティ設定を行った STREAM ベンチマーク

STREAM は大きさが n の実数ベクトル $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \boldsymbol{R}^n$ に対する 4 種類の演算 Copy, Scale, Add, Triad に要する実行時間を用いたメモリ帯域幅に対するベンチマークである. なおいずれのベクトルも倍精度浮動小数点型 (`double` 型) の大きさ n の配列として, scalar は実数変数 (`3.0`) として実装されている.

表 4 4-way Sandybridge-EP 上のアフィニティ設定毎のスレッド数と使用する NUMA ノード数, NUMA ノード内の物理コア数, 物理コア上で同時実行するスレッド数

(a) Scatter アフィニティ

スレッド数	1	2	4	8	16	32	64
NUMA ノード数	1	2	4	4	4	4	4
NUMA ノード内の物理コア数	1	1	1	2	4	8	8
物理コア上のスレッド数	1	1	1	1	1	1	2

(b) Compact アフィニティ

スレッド数	1	2	4	8	16	32	64
NUMA ノード数	1	1	1	1	1	2	4
NUMA ノード内の物理コア数	1	2	4	8	8	8	8
物理コア上のスレッド数	1	1	1	1	2	2	2

(c) Compact+ アフィニティ

スレッド数	1	2	4	8	16	32	64
NUMA ノード数	1	1	1	1	2	4	4
NUMA ノード内の物理コア数	1	2	4	8	8	8	8
物理コア上のスレッド数	1	1	1	1	1	1	2

$$
\begin{aligned}
&\textbf{Copy} && c_i = && a_i, && (i = 0, 1, ..., n-1)\\
&\textbf{Scale} && b_i = && \mathrm{scalar} \cdot c_i, && (i = 0, 1, ..., n-1)\\
&\textbf{Add} && c_i = && a_i + b_i, && (i = 0, 1, ..., n-1)\\
&\textbf{Triad} && a_i = && b_i + \mathrm{scalar} \cdot c_i, && (i = 0, 1, ..., n-1)
\end{aligned}
$$

まず, 図 5 に C/C++ 言語上で malloc/new 関数を用いてメモリ確保を行った Triad 演算の実装例を示す. この場合, 図 2 で示したように, 特定のローカルメモリ上に固定するような制御は行っていないため, スレッドに応じてアクセスコストの異なる不均質なメモリアクセスが多発することが予測される. しかしながらメモリ確保はファーストタッチに基づいて確保されるため, 初期化時のスレッドと同じスレッドで演算を行うことができればローカルなメモリアクセスとなる.

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

図 5 OpenMP を用いたスレッド並列 Triad 演算 (アフィニティ設定なし w/o)

続いて, 図 6 は ULIBC の関数 (図 7) を用いて, アフィニティ設定とローカルメモリへのメモリ確保設定を行った Triad 演算である. まず (1) `set_affinity_policy` 関数でアフィニティ設定 `{COMPACT, COMPACT_PLUS, SCATTER}` のいずれかを指定し, (2) `get_online_numa_nodes` 関数で取得したスレッド並列時に実行される NUMA ノード数分の固定されたローカルメモリ領域を (8) `lmalloc` 関数を用いて動的メモリ確保する. その後, スレッド並列領域の先頭で OpenMP の `omp_get_thread_num` 関数を用いてスレッド ID を取得し, アフィニティ設定で割り当てられる Processor ID, Package ID, NUMA ノード内の各論理コアに割当表での割り当て順に与えた通し ID をそれぞれ (3) `get_numa_procid` 関数, (4) `get_numa_nodeid` 関数, (7) `get_numa_ordcoreid` 関数を用いて取得する. 各スレッドは得られた Processor ID で指定した論理コア上に (9) `set_affinity` 関数を用いて固定する. その後, 各 NUMA ノード上に配置されたスレッド数を (2) `get_online_numa_cores` 関数を用いて取得し, 各スレッドが担当する配列の範囲を計算して, 各スレッドは割り当てられた Triad 演算を行う. また, スレッド計算後には (10) `clear_affinity` 関数を用いて, プログラムの実行開始時に取得したデフォルトのアフィニティ設定で現在のアフィニティ設定を上書きし, 以後の処理に現在のアフィニティ設定が影響が出ないように初期化する. また, この例では使用していない (5) `get_numa_coreid` 関数や (6) `get_numa_smtid` 関数は, それぞれ スレッド ID に対応する物理コア ID (Core ID) や, 物理コア上で実行されるスレッド ID (SMT ID) を取得することができる.

```
long N[MAX_NODES];
double *A[MAX_NODES], *B[MAX_NODES], *C[MAX_NODES];

set_affinity_policy(SCATTER);
for (int k = 0; k < get_online_numa_nodes(); ++k) {
  A[k] = lmalloc(sizeof(double) * N[k], k);
  B[k] = lmalloc(sizeof(double) * N[k], k);
  C[k] = lmalloc(sizeof(double) * N[k], k);
}

#pragma omp parallel
{
  int id = omp_get_thread_num();
  int procid = get_numa_procid(id);
  set_affinity(procid);

  int nodeid = get_numa_nodeid(id);
  int coreid = get_numa_ordcoreid(id);
  int lnp = get_numa_online_cores(nodeid);

  double *lA = A[nodeid], *lB = B[nodeid], *lC = C[nodeid];
  const long q = N[nodeid] / lnp, r = N[nodeid] % lnp;
  const long ls = q * (coreid+0) + min(r, coreid+0);
  const long le = q * (coreid+1) + min(r, coreid+1);

  for (long j = ls; j < le; ++j) {
    lA[i] = lB[i] + scalar * lC[i];
  }
  clear_affinity();
}
```

図 6 ULIBC による Scatter アフィニティ設定を適用した STREAM ベンチマークの Triad 演算

図 8 に, 以上のように実装した STREAM ベンチマークの Triad 演算に対する 4 種類アフィニティ設定 w/o (アフィニティ設定なし), Scatter, Compact, Compact+ 毎の実行結果 (メモリ帯域幅 GB/s) をまとめる. 要素数は 2^{27} とし, 用いた計算機環境は 4-way Sandybridge-EP である. Scattar では, 1, 2, 4 とスレッド数の増加に従い使用する CPU ソケットも 1, 2, 4 と増加する. 続いて 4, 8, 16, 32 とスレッド数を変化させると, 各ソケットに 1, 2, 4, 8 コ

```
(1) int set_affinity_policy(int policy)
(2) int get_online_numa_nodes(void)
(3) int get_numa_procid(int thread_id)
(4) int get_numa_nodeid(int thread_id)
(5) int get_numa_coreid(int thread_id)
(6) int get_numa_smtid(int thread_id)
(7) int get_numa_ordcoreid(int thread_id)
(8) void *lmalloc(size_t sz, int node_id)
(9) int set_affinity(int processor_id)
(10)void clear_affinity(void)
```

図 7 ULIBC で提供する関数群

アずつ使用した 4 ソケット実行となる．一方, Compact, Compact+ では, 1, 2, 4, 8 とスレッド数の増加に従い, 1, 2, 4, 8 コアを使用した 1 ソケット実行となる．その後, Compact では 16 スレッド時に各物理コア上に 2 スレッドずつ固定した配置となり, 32, 64 とスレッド数の増加に従い, 2, 4 ソケット実行となる．また, Compact+ では 16, 32 スレッド時には各物理コアに 1 つずつ配置した 2, 4 ソケット実行となる．いずれも 64 スレッド時には, 各物理コア上に 2 スレッドずつ固定した配置 (Hyper-Threading 機構) と同じ配置となる．以上より, 使用するソケットが最大になるようにコア配置を行う Scatter が最も高いメモリ帯域幅を示し, 続いて, Compact+, Compact となった．このように, メモリバンド幅に対しては有効とはならなかったものの, 各スレッド間に頻繁な通信が必要となる場合には, Compact もしくは Compact+ による性能向上に期待できる．また, アフィニティ設定を行わなった場合は, 32, 64 スレッド時で性能低下が起きるが, ULIBC によるアフィニティ設定を行うことで, メモリアクセスの不均一化による性能低下を抑えていることが確認できる．

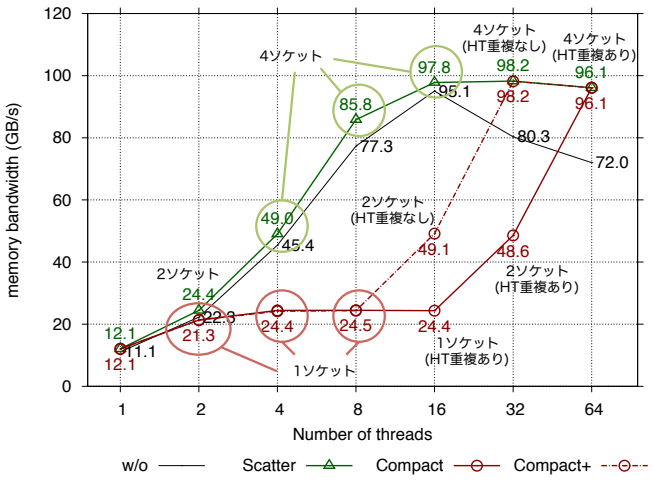


図 8 NUMA を考慮した STREAM ベンチマーク

4. ULIBC を用いた NUMA を考慮した共有メモリ型並列アルゴリズムの高速化

4.1 幅優先探索

近年, グラフを用いた解析手法は様々な分野で盛んに研究されており, 中でも幅優先探索 (Breath-first search; BFS) は基本的かつ重要なグラフ処理といえる．並列 BFS アルゴリズムである Beamer's algorithm [8] は, 探索済の点集合から未探索の隣接点集合を発見するという従来手法である Top-down 探索と, 未探索の隣接点集合から探索済みの点集合を発見する Bottom-up 探索を組み合わせ, 点数 2^{28}, 枝数 2^{32} からなる Kronecker graph に対し 4-way Intel Xeon E7-8870 上で高い性能 5.1 GTEPS を達成している．ここで GTEPS は 1 秒あたり 10^9 枝を探索する性能を表す．さらに著者らは Scatter アフィニティを基本とした NUMA を考慮した高速化を行い, 同等の計算機環境 4-way Sandybridge-EP 上で 2.2 倍となる 11.15 GTEPS を達成した [9]．なお本論文では, 文献 [9] では取り扱わなかったアフィニティ毎での性能, 特性について議論する．

図 9 に, SCALE=26, edgefactor=16 とした点数 $n = 2^{26}$, 枝数 $m = 2^{30}$ からなる Kronecker graph に対する BFS 計算性能 (GTEPS) をまとめたものである．我々のアルゴリズムは実行時間の大部分を各 NUMA ノードに分割した領域へのローカルなメモリアクセスで計算できるように局所性の改善を行っており, 使用するスレッド数の増加に従う性能向上が得られる．また, 16 スレッド時のアフィニティ設定毎の NUMA ノード数は Compact : Compact+ : Scatter = 1 : 2 : 4 であるが, その際の性能差は Compact : Compact+ : Scatter = 1.00 : 1.47 : 1.87 となることから, 同一のスレッド数であれば NUMA ノード数が多い割当の方が高い性能となることが確認される．このように, STREAM ベンチマークの Triad 演算に似た性能特性を確認できるものの, 参照や書出し対象のデータ量が大きくかつ複雑アクセスパターンとなるため, ファーストタッチでのメモリ確保が有効とならず, アフィニティ設定を行わない w/o が極端に低速となる．

4.2 閉路を持たないパスの数え上げ

Knuth の Simpath アルゴリズム [12] は, 与えられたグラフ $G = (V, E)$ 上の 2 点 $s, t \in V$ 間の閉路を持たないパスを効率的に列挙する．ここで, 枝集合 E は $\{e_1, e_2, ..., e_m\}$, 点集合 V は $\{s, v_1, v_2, ..., v_\ell, t\}$ と与えられ, 各枝の添字は始点 s からの幅優先探索で決定するものとする．図 10 (a) のようなグラフ上で s-t 間の閉路を持たないパスの列挙を行う際, Simpath アルゴリズムは各枝 e_i をパスに含めるか否かを表現した図 10 (c) の DD (Decision Diagram; 決定グラフ) を構築する．ここで, DD 内の各節点 e_i が持つ実線矢印と破線矢印はそれぞれ 1-arc と 0-arc と呼び, e_i をパス

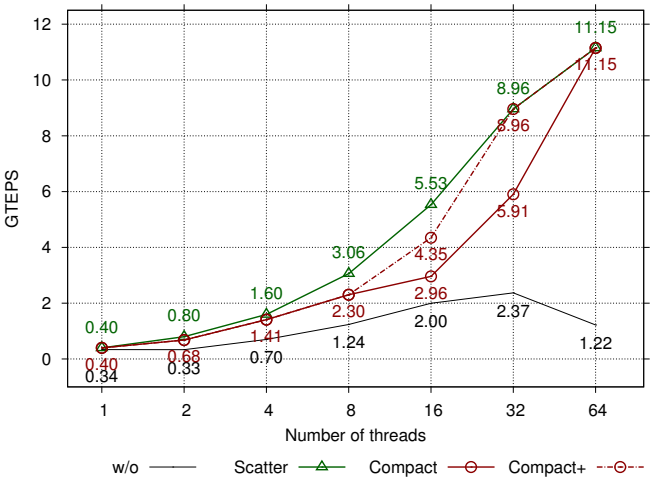


図 9 graph500 に対するアフィニティ毎の TEPS 値

に含めるか否かを表す. 各 e_i の選択が G における正しいパスを構成する場合は DD の端点が 1 (1-terminal), そうでなければ 0 (0-terminal) となるため, 閉路を持たないパスの総数は端点が 1 となる組合せを数えれば良い. 図 10 (a) の例では, 端点が 1 となるのは $\{e_1, e_4\}$ と $\{e_2, e_3, e_4\}$ となり, 閉路を持たないパスの総数は 2 となる.

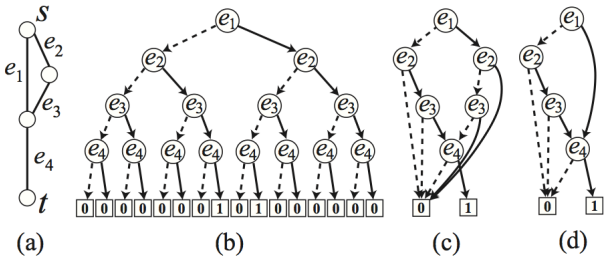


図 10 ZDD と Simpath アルゴリズム

Simpath アルゴリズムは始点 s からトップダウンに幅優先探索で DD を構築していく際に, (1) 等価な節点を共有する, (2) 節点を含めると条件を満たさない場合その節点を削除して 0-terminal に接続する, といった ZDD (zero-suppressed binary decision diagram) [11] の簡約化規則を用いる. 最終的には図 10 (d) のような規約な ZDD に変換することができる. Simpath は効率的ではあるものの逐次アルゴリズムであるため, 現在のマルチコアプロセッサの性能を引き出すことは難しい.

すでに著者らは Simpath に対する 3 種類並列アルゴリズム Node-based, Range-based, Edge-based を提案し, 中でもロックフリーアルゴリズムの Edge-based approach (EBA) は高速かつ並列効率が高く, 32 スレッド並列実行時の EBA は 逐次アルゴリズムの 7 倍の性能となる [13]. EBA におけるスレッド数を k とした並列計算では. スレッド j は各レベル $i \bmod k$ $(0 \leq i \leq m)$ に対応した FIFO キュー N_i の情報を保持する. スレッド j は FIFO キュー N_i から節点 n_i をデキューし, 枝 e_i を選択するか否かに対応する新たな節点 n_i^0 と n_i^1 を生成し, それぞれ $l = (i+1) \bmod k$ となるスレッド l に移動する. なお, スレッド l における n_i^0 と n_i^1 の重複確認は, スレッドローカルなハッシュテーブルを参照しロックフリーで処理される.

続いて, 本研究により ULIBC を用いた NUMA アーキテクチャを考慮した高速化について説明を行う. EBA におけるスレッド j は自身が保持している FIFO キュー $N_{i \bmod k}$ へのデータ参照と, スレッド l が保持している FIFO キュー $N_{(i+1) \bmod k}$ へのデータ書込みが頻繁に行われる (図 11). そのため隣接のスレッドの距離が短くなるように, 論理コアへ割り当てることが重要となる.

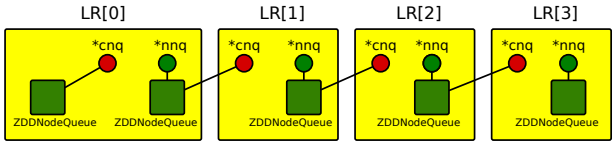


図 11 Edge-based parallel Simpath のメモリアクセスパターン

図 12 は, 16×16 の格子グラフ (点数 289, 枝数 544) 上で EBA を用いて経路数の列挙に要した時間を各アフィニティ設定 w/o (アフィニティ設定なし), Scatter, Compact, Compact+ 毎にまとめる. なお, 計算機環境は 4-way Sandybridge-EP である. まず w/o は並列数の増加に従い性能向上しており, アフィニティ設定を行わずとも並列効果が高いことが確認できる. 一方で, 1 スレッド時に w/o は 1478 秒要したが, Scatter, Compact, Compact+ はそれぞれ 752 秒となり, アフィニティ設定による安定性の向上が確認できる. Scatter は, 1 スレッド時と比べて 2 スレッド時では隣接スレッドの距離の増大による一時的に性能低下するが, さらなる並列数の増加に従い性能向上し, 最も高速な 64 スレッド時には w/o よりも高い性能となる. 一方, Compact と Compact+ は 1, 2, 4, 8 と並列数の増加に従い性能向上するものの, Compact では 32 スレッド時に, Compact+ では 16 スレッド時に, 一時的に性能低下する. このとき, それまで割り当てが 1 ソケット内に限定していたのに対して, 2 ソケットに横断した割当となっている. 16 スレッド時に Compact と Compact+ の性能差はほぼ倍となることから, 各物理コア上での 2 つ論理コアの同時実行 (Hyper-Threading 機構) よりも CPU ソケットの横断によるメモリアクセスの不均一化によるオーバヘッドが大きいことが確認できる. さらに 32 スレッド時での Compact と Compact+ はほぼ同等であるため, 物理コアを 16 ずつ使用した 2 ソケットの性能と, 論理コアを 32 使用した 1 ソケットの性能が同等であることも推測できる. Compact, Compact+ の 64 スレッド時には w/o (85.8 秒) と比べて, 49.2 秒と 1.74 倍高速化に成功した.

続いて図 13 に, 格子グラフの大きさを 10 × 10 から 18 × 18 と変化させた際の, アフィニティ設定毎の 64 ス

レッド時の実行時間をまとめる．いずれのサイズに対しても Compact は w/o よりも同等かより高い性能を示している．

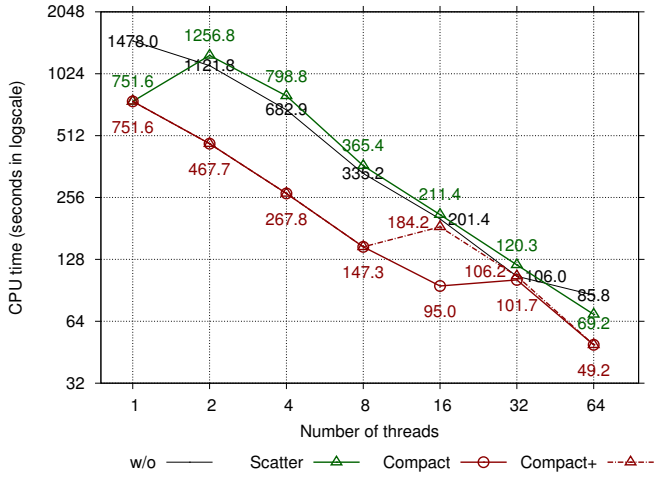


図 12 16 × 16–格子グラフに対するアフィニティ設定毎の実行時間

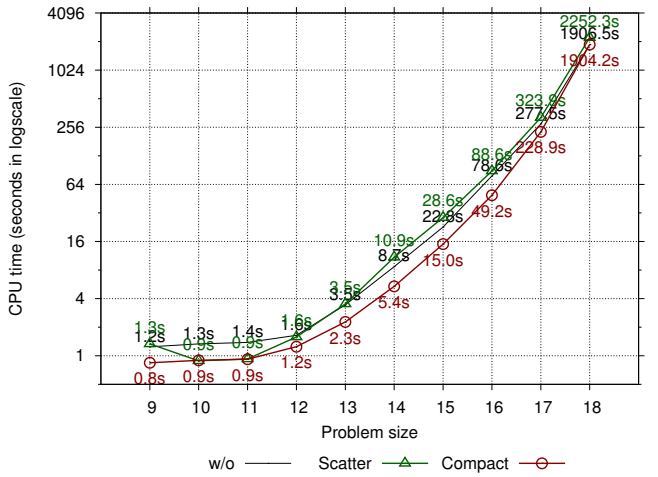


図 13 格子グラフに対するアフィニティ設定毎の EBA の実行時間

4.3 SDPA

半正定値計画問題 (Semidefinite programming; SDP) は組合せ最適化, システムと制御, データ科学, 金融工学, 量子化学など非常に幅広い応用を持ち, 現在最適化の研究分野で最も注目されている最適化問題の一つとなっている．また今後のエネルギー供給計画（スマートグリッド等）では非線形の複雑な最適化問題を扱う必要があり, これらの問題に対して強力な緩和値を算出できる SDP の高速計算技術の確立が急務とされている．SDP に対しては高速かつ安定した反復解法である内点法アルゴリズムが存在しているが, 巨大な線形方程式系の計算が大きなボトルネックとなっている．具体的には線形方程式系の行列要素の計算 (以下 ELEMENTS) と行列の Cholesky 分解 (以下 CHOLESKY) である．著者らのグループでは内点法アルゴリズムを記述したソフトウェアの開発・評価・公開を 15 年以上行っており, 疎性の追求, 計算量やデータ移動量などによる計算方法の自動選択などの技術を他に先駆けて実現し, 大規模な並列計算等によって上記のボトルネックの高速化と世界最大規模の SDP を高速に解くことに成功している [14]. 特に CHOLESKY に関しては東京工業大学のスーパーコンピュータ TSUBAME 2.0 上において, 多数 GPU の活用や計算と通信のオーバーラップ技術を応用することによって, 制約式の数が 148 万以上となる世界最大規模の巨大 SDP を解き SDP の世界記録の更新及び最大で 533 TFlops (Cholesky 分解: 4080 個の NVIDIA Tesla M2050) の性能を達成した [15]．その後, TSUBAME 2.5 において制約式の数が 233 万以上となる巨大 SDP を解き, 更なる SDP の世界記録の更新及び最大で 1.713 PFLOPS (Cholesky 分解: 4080 個の NVIDIA Tesla K20X) の性能を達成した [16]．一方, ELEMENTS に関しては入力問題の疎性の活用や行列要素の並列計算によって大幅な高速化に成功しているが, メモリバンド幅に律速されているため, 浮動小数点演算律速の CHOLESKY のように GPU 等による加速は容易ではない．文献 [16] においては分散メモリ環境上でのアフィニティ設定とメモリインターリーブ設定による ELEMENTS を提案しているが, 今回は著者らが開発した 1 ノード版の SDP 用のソフトウェア SDPA [14] を用いて, 図 14 に示す ELEMENTS 計算の性能向上に関する検証実験を行う．

```
set_affinity_policy(policy)
B = ∅
for l = 1, 2, ··· , h do
  for j ∈ {j | F_j^l ≠ ∅} parallel(thread) do
    set_affinity(omp_get_thread_num())
    for i ∈ {i | F_i^l ≠ ∅} do
      Selects F_1, F_2 or F_3 and compute B_ij^l
      B_ij = B_ij + B_ij^l
    end
  end
end
clear_affinity()
```

図 14 線形方程式系の行列要素 $\boldsymbol{B}_{ij}$ の計算 (ELEMENTS)

図 15, 16 に, Intel Xeon E7-4870 が 4 基搭載された共有メモリ型計算機環境 4-way Westmere-EX 上の 2 種類の SDP 問題 mater-6, tensor4441 の実行時間を示す．mater-6, tensor4441 ともに疎性を有し, ELEMENTS がボトルネックとなっている．まず tensor4441 はスレッド数の増加に従い一定の性能向上が得られているが, w/o や Scatter では 2, 4, 8 スレッド時が同等でそれ以降は性能低下してしまうのに対して, Compact では 8 スレッドまで性能向上を確認することができる．一方, mater-6 はスレッド数の増

加に従い実行時間が増加してしまい, 最も性能が高いのはスレッド数を 1 とした逐次実行時である. また, スレッド数の増加による性能低下の影響を受けにくいのは Compact であることも確認できる. この問題に関しては今後とも注意深く観察する必要がある. 以上のように, 計算機実験において特性異なるどちらの問題に対しても, ソケットを横断するデータアクセスがオーバヘッドになっていることが確認された.

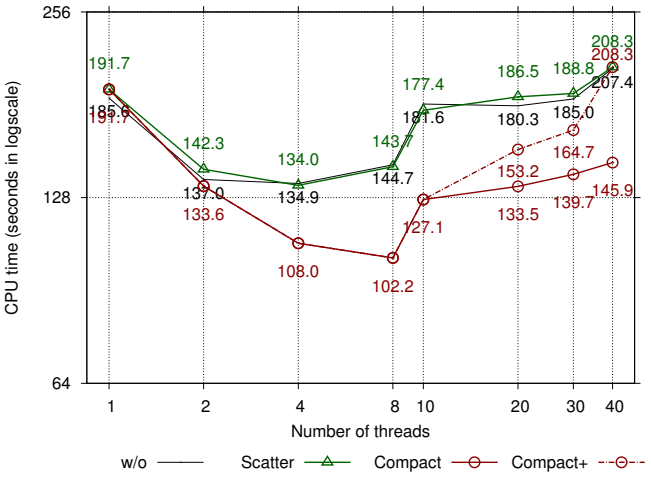


図 15 SDP 問題 tensor4441 に対するアフィニティ毎の実行時間

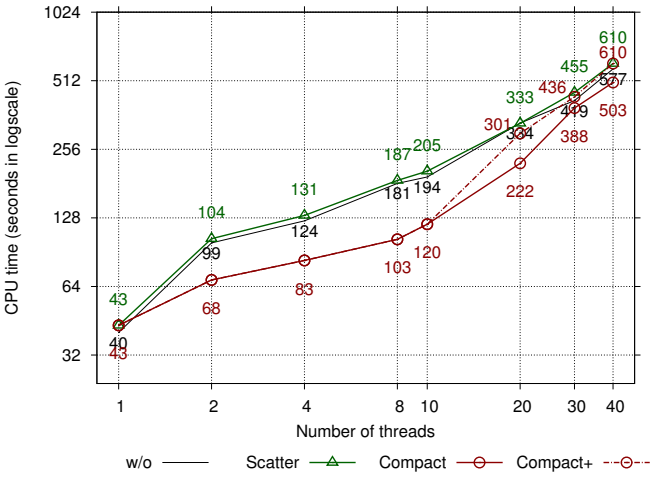


図 16 SDP 問題 mater-6 に対するアフィニティ毎の実行時間

5. まとめ

本研究では CPU アフィニティ設定やローカルメモリ上に固定したメモリ領域確保を行うためのライブラリ ULIBC を開発し, 性質の異なる 4 種類のアプリケーションに対する高速化を行い, いずれに対しても性能改善を達成した. ULIBC は 3 種類のアフィニティ設定 Scatter, Compact, Compact+ に対応し, スレッド並列領域においては計算機トポロジを参照することで, 各スレッドが配置している論理コアが計算機上のどの箇所に配置されているか容易に把握することができる. これらの情報は NUMA アーキテクチャに対する高速化を行う上で非常に重要で, 他のライブラリでは提供されていない機能である. 例えば, 我々が文献 [9] で行ったように NUMA を意識しメモリアクセスの局所性を高めたアルゴリズムを設計することができた場合には, 既存のライブラリや OS が提供する関数群を用いることで実現することは難しくない. しかしながら論文中でも行っているように Simpath や SDPA の ELEMENTS 演算などのより複雑なメモリアクセスを行うアルゴリズムに対する高速化においては, アルゴリズム上の性質上, どのアフィニティ設定が有効なのか, またその際にどれほどのボトルネックが存在しているかなどの解析が必要となる. そういった解析にはアフィニティ設定やメモリ領域確保の挙動の容易に把握できる環境が必要である. 今後も引き続き ULIBC をベースとした高速なアルゴリズムの開発を行う予定である.

謝辞 本研究の一部は, 科学技術振興機構 CREST「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」, ならびに科学技術振興機構 ERATO 湊離散構造処理系プロジェクトの助成を受けたものである.

参考文献

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader: Scalable graph exploration on multicore processors, Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC10), IEEE Computer Society, 2010.

[2] 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. In Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010), 2010.

[3] J. Treibig, G. Hager, G. Wellein: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. PSTI2010, 2010.

[4] Intel(R) C++ Compiler XE 13.1 User and Reference Guide, Intel Thread Affinity Interface.

[5] L. Huang, H. Jin, L. Yi, and B. Chapman: Enabling locality-aware computations in OpenMP, Journal Scientific Programming - Exploring Languages for Expressing Medium to Massive On-Chip Parallelism archive Vol. 18, Issue 3-4, pp. 169–181, 2010.

[6] 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. Japan, vol. 54, no. 4, pp. 259–280, 2011.

[7] M. Frasca, K. Madduri, and P. Raghavan: NUMA-Aware Graph Mining Techniques for Performance and Energy Efficiency, Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC12), IEEE Computer Society, 2012.

[8] S. Beamer, K. Asanović, and D. A. Patterson: Direction-optimizing breadth-first search, Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Stor-

age and Analysis (SC12), IEEE Computer Society, 2012.
[9] Y. Yasui, K. Fujisawa, and K. Goto: NUMA-optimized Parallel Breadth-first Search on Multicore Single-node System, The proceedings of the IEEE BigData 2013, Santa Clara, USA, IEEE Computer Society, 2013.
[10] S. Kuo: Intel(R) 64 Architecture Processor Topology Enumeration, Intel Corporation White paper, 2012.
[11] S. Minato: Zero-suppressed BDDs for set manipulation in combinatorial problems, DAC '93: Proc. 30th Int. Conf. Design automation, 1993.
[12] D. E. Knuth: The Art of Computer Programming, Vol. 4, Fascicle 1, Addison-Wesley, pp.121–123, 2009.
[13] S. Takeuchi, J. Kawahara, A. Kishimoto, and S. Minato: Shared-Memory Parallel Frontier-Based Search, Algorithms and Computation Lecture Notes in Computer Science Volume 7748 (WALCOM 2013), 2013.
[14] K. Fujisawa, K. Nakata, M. Yamashita, M. Fukuda: SDPA Project: Solving large-scale semidefinite programs. J. Oper. Res. Soc. Japan, **50**, 278–298, (2007)
[15] K. Fujisawa, T. Endo, H. Sato, M. Yamashita, S. Matsuoka and M. Nakata: High-Performance General Solver for Extremely Large-scale Semidefinite Programming Problems, Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC12), IEEE Computer Society, 2012.
[16] 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. ACM/IEEE Int. Conf. Parallel & Distributed Processing Symposium (IPDPS 2014), IEEE Computer Society, 2014.