SC16 参加報告

東京大学情報基盤センターの教員が 2016 年 11 月 13 日から 18 日までアメリカ合衆国ユタ州ソルトレイクシティにて開催された SC16 (The International Conference for High Performance Computing, Networking, Storage and Analysis) に参加した。本会議は、高性能計算 (HPC) 分野では著名な国際会議であるとともに、様々な情報技術関連企業や研究所・大学等の技術展示会でもある。本稿はその参加報告として、SC16 に参加した本学教員が現地で見聞きした中から気になったことなどを記す。

はじめに

SC16 の会場となったソルトレイクシティはアメリカ合衆国ユタ州の中央部にある州都である。言わずと知れた 2002 年冬季五輪の会場となった都市であり、SC16 が開催された Salt Palace Convention Center は五輪の際にメディアセンターとして用いられた建物である。最近では SC12 にもまったく同じ会場で SC が行われている。SC12 では開催前日に雪が積もり銀景色の SC となったが、今回は後半になって多少の積雪があった程度であり、前回の経験から防寒具をしっかりもっていった参加者達には拍子抜けであった。交通については、SC12 の時点でも市内を走る Light rail が便利であったが、今回は空港への延伸が完了しており、空港と会場、会場とホテル、市内のアクセスはいずれも快適であった。ただし会場や Light rail 停留所から遠いホテルで繰り返しミーティングを行った参加者らも少なからず居たようで、苦労したという声も聞かれた。

図 1　SC16 会場 (Calvin L. Rampton Salt Palace Convention Center)

SC-XY について

本会議は以前は Supercomputing-XY(XY：開催年) という名称が用いられており、1997 年に SC-XY という名称に変更された。1988 年フロリダ州オーランドで第 1 回が開催されてから、毎年 11 月にアメリカ各地を転々としながら開催されており、今回は Supercomputing-88 から数えて 28 回目である。

本会議は、基調講演、研究発表、パネル討論、BoF(Birds of a Feather: 特定のトピックを定めた小規模集会)、主要技術の理解を助けるチュートリアル、併設される多数のワークショップなどで構成されている。研究発表については 81 件行われ、また 12 件のパネル、138 件のポスター、37 件のチュートリアル、37 件のワークショップ、12 件の招待講演、55 件の BoF など盛りだくさんの内容であった。また、企業や各種研究機関による最新の製品や技術の展示発表も注目すべき内容である。

主催者発表によると、SC16 の来場者数 (参加登録者数) は 11,100 人を越えた。また展示については 349 団体が参加し、過去最大の展示面積となった。初回参加の団体は 44 あり、また米国外からは 25 カ国 120 団体が参加した。

東京大学情報基盤センターによる展示

東京大学情報基盤センターは「ITC/JCAHPC, The University of Tokyo」という名義によるブース展示を行った。

今回は、例年同様に情報基盤センターの提供する計算資源（スーパーコンピュータなど）や研究事例を紹介するポスターの展示を行うのに加えて、ブース名に「JCAHPC」と入っていることからわかるように、筑波大学計算科学研究センターと共同で設立した最先端共同 HPC 基盤施設 (JCAHPC) にフォーカスした展示も実施した。今回は筑波大学計算科学研究センターによる「CCS/JCAHPC, University of Tsukuba」ブースと通路を挟んで隣り合わせのブースを確保できたため、ブースの色調を合わせるなど一体感を演出した。両ブースには最新のスーパーコンピュータシステム Oakforest-PACS のノードとシャーシを静態展示し、関係するパンフレット・チラシ・グッズの配布を行った。また恒例となっているブースでのプレゼンテーションも両ブースで協力して実施した。後述のように Oakforest-PACS が TOP500 等のランキングにて上位入賞したこともあり、多くの来場者が両ブースを訪れた。

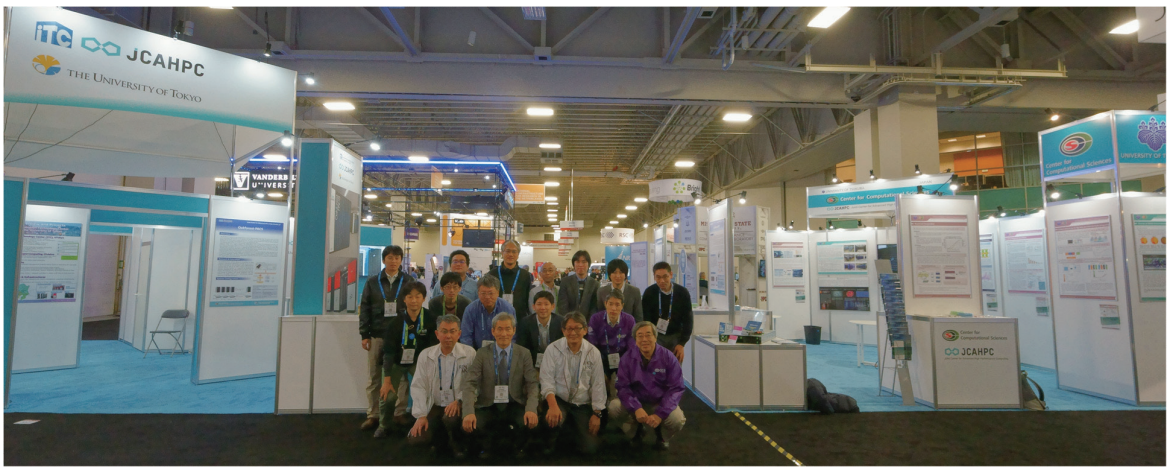

図 2　ブース展示の様子 1（ゲストによるブースプレゼンテーション、展示物を紹介する様子、「ITC/JCAHPC, The University of Tokyo」ブースの集合写真、東京大学情報基盤センターと筑波大学計算科学研究センター合同での集合写真）

各種のランキングについて

毎年の SC ではスーパーコンピュータの性能に関する様々なランキングが更新される。今回は JCAHPC にて 12 月に正式運用を開始した Oakforest-PACS に関する情報を中心に紹介する。

Top500 (http://www.top500.org/) は世界のスーパーコンピュータの性能を LINPACK という係数行列が密の連立一次元方程式を解くベンチマークの処理速度によって競うものである。1993 年の開始以来、6 月にヨーロッパで行われる会議である ISC と、本会議 SC にて年 2 回の更新を続けている。しかしここ数回の Top500 はランキング上位の変化が乏しく、特に Top5 は第 41 回 (2013 年 6 月) から前回までまったく入れ替わりがない状態であった。今回は、Top4 システムこそ前回と変わらぬ状態であったものの、5 位に米国 NERSC の Cori、6 位に JCAHPC の Oakforest-PACS がランクインした。これら 2 システムは Intel 社の最新世代の Xeon Phi メニーコアプロセッサ (コードネーム Knights Landing) を搭載したシステムである。また日本のシステムにのみ注目すると、2011 年 6 月から日本で一番高速なシステムであった「京」コンピュータがついに首位から退いたことになる。

今回の Top500 でも中国の勢いは強く、前回の 109 システムを大きく上回る 171 システムがランクインした。一方、前回 199 システムをランクインさせていた米国は中国と同数の 171 システムへと減少した。また日本のランクイン数は前回の 37 システムからさらに減少し、27 システムであった。ただし Top100 に限って見れば、米国の 40 システムに対して中国は 8 システム、日本も 10 システムがランクインしている。超大規模システムについては、今回も米国の強さが際立っていると言える。

なお今回日本から新規にランクインしたシステムは、

1. JCAHPC の Oakforest-PACS システム (6 位)
2. 京都大学学術情報メディアセンターの Camphor 2 システム (33 位)
3. 東京大学情報基盤センターの Reedbush-U システム (361 位)

以上 3 システムであった。ちなみに Oakleaf-FX システムは 104 位まで後退しており、1PFLOPS を越えるシステムは 117 を数えている。

Top500 の結果を元に電力当たりの LINPACK 性能を比較したランキングとして Green500 (http://www.gree500.org/) も実施されている。近年の Green500 では日本の Pezy Computing 社によるシステムが 3 連覇していたが、今回は NVIDIA 社による最新 GPU(Pascal アーキテクチャ) を搭載した DGX SATURNV システムが首位を獲得した。このシステムは最新 GPU である Tesla P100 を大量に搭載したシステムであり、Top500 では 3.307PFLOPS で 28 位にランクインしている。2 位の Piz Daint も同様に P100 を搭載したシステムであり、こちらは 9.779PFLOPS、Top500 では 8 位である。DGX SATURNV システムは電力あたり性能が 9462.1 MFLOPS/W と非常に高い。仮にこのシステムをさらに多く並べた場合、計算上では 1ExaFlops が 106MW で達成可能であり、現実的な消費電力 (40MW 程度) による ExaFlops 到達の可能性が見えてきたと言える。なお、前回 1 位であった理化学研究所の Shoubu(菖蒲) は 3 位であった。

以上のように、今回の Green500 では NVIDA 社の GPU がその消費電力あたり性能の高さを見せつけた格好となった。その一方で、Intel 社のメニーコアプロセッサを搭載したシステムも、ドイツ国内で富士通社が納入した QPACE3 システムが 5 位 (5806.3MFLOPS/W)、

Oakforest-PACS が 6 位 (4985.7MFLOPS/W) など、Top10 のうち 5 システムを占めており、大きな存在感を見せていた。

Top500 はスパコンのランキングとして広く知られたランキングであるが、LINPACK で高性能を得るためには大規模な問題を解く必要がある。そのため大規模システムで高い LINPACK スコアを得るためには数十時間もの実行時間が必要なこともあり、実行の負担が大きいことが問題となっている。さらに LINPACK が行う計算の内容はスパコン上で稼働している実アプリケーションと大きな隔たりがあるため、より実アプリケーションに近く実行する負担の小さいベンチマークが必要であるという議論が行われてきた。HPCG (High Performance Conjugate Gradients, https://software.sandia.gov/hpcg/) はそのような背景から 2013 年に提案された新しいベンチマークである。HPCG は有限要素法から得られる疎行列を対象とした線形ソルバーであり、大規模問題向けのマルチグリッド法を前処理として採用しており、LINPACK よりも実アプリケーションに近いと考えられる。また最低 30 分計算を実行して解が収束していることを示せば良いため、短時間での測定が可能である。

HPCG も 2014 年からは ISC および SC にてランキングが更新されている。前回 (2016 年 6 月) のランキングでは 1 位が Tianhe-2、2 位が「京」コンピュータ、3 位が Sunway TaihuLight という順序であったが、今回は「京」コンピュータがさらなる最適化によって初めて 1 位を獲得した。また Oakforest-PACS も Sunway TaihuLight を越えて 3 位にランクインした。HPCG は LINPACK と比べて 1% から 3% 程度しか性能が出ないことが多い問題であるが、Top500 7 位の「京」コンピュータは 5.30% という高い性能を達成しており、また Top500 1 位の Sunway TaihuLight は 0.40% 未満という低い性能に留まっている。このように HPCG は Top500 (LINPACK) とは毛色の異なるランキングとなっていることがわかる。HPCG のさらなる内容や Oakforest-PACS の 3 位入賞についての情報は既に本部門の Web ページに掲載されているため、そちらも参照されたい [1] [2]。

大規模グラフの解析に関する性能を競う Graph500 (http://www.graph500.org/) では Top5 システムが前回 6 月から変動無しであった。Graph500 はその難しさから Top500 よりも遅れて最適化が進む傾向があるため、次回以降に Knights Landing や Pascal がどれだけ上位にランクインしてくるか気になるところである。

1　Oakforest-PACS リリースアナウンス
http://www.cc.u-tokyo.ac.jp/system/ofp/release-20161118.html

2　HPCG についてのさらに詳しい資料
http://www.cc.u-tokyo.ac.jp/system/ofp/KN_HPCG-3.pdf

図 3　HPCG BoF における表彰の様子

メイントラック論文について

An Ephemeral Burst-Buffer File System for Scientific Applications by Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu

Florida State University と Lawrence Livermore National Laboratory の著者らによる ``An Ephemeral Burst-Buffer File System for Scientific Applications" と題された論文。Burst-Buffer とは、アプリケーションとファイルシステムの間に入り、I/O を高速化する機構のことである。実際に専用のノードをサーバとして配置する場合もあるし、クライアントノードに配置される場合もある。クライアントが直接ファイルシステムへ I/O を発行する場合と比べ、(1) 書き込みをローカルもしくは近くにあるノードで吸収する、(2) ファイルシステムが苦手とするアクタスパターン（例 : 多数の小さな I/O）を整形してファイルシステムに発行することでアクセスを高速化する、などの効果がある。本研究の Burst-Buffer ファイルシステム（BurstFS）はクライアントノードに配置されるもので、各計算ノードのローカル SSD を用いる。書き込みの際、データはすべてローカル SSD に書き込まれる。メタデータ（ファイルとそのオフセットから、現在それを保持するノードを返すデータ構造）は、分散キーバリューストア（MDHIM）を用いて管理される。読み込みの際はこのメタデータを参照する。読み込む範囲（オフセットの範囲）に対して範囲検索（range query）を発行して、データの位置を求める。MDHIM は従来範囲検索をサポートしていないが、本研究ではそのために MDHIM を拡張している。N-1 write（多数のプロセスが 1 ファイルへ書き込むパターン）で 1024 プロセス（64 ノード）までほぼ線形に書き込み速度がスケールし、並列ファイルシステム OrangeFS へ直接読み書きを行った場合との比較では、8 倍程度高速化している。

本センターで運用する Oakforest-PACS でも Burst-Buffer が導入される。その性能特性などについても今後報告していく予定である。

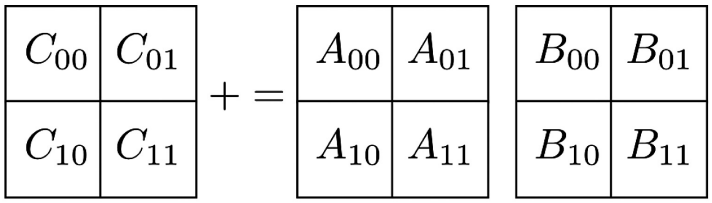

図 4　小行列への分解

Strassen's Algorithm Reloaded by Jianyu Huang, Tyler M. Smith, Greg M. Henry, Robert A. van de Geijn

The University of Texas at Austin の著者による、``Strassen's Algorithm Reloaded" と題する論文。

二つ n × n 行列 A, B の密行列積（正確には、C += A B）は、通常Θ (n3) の計算コストを要するが、Strassen のアルゴリズムはこれをΘ (nlog 7) ≈ Θ (n2.807) で実行するアルゴリズムで、1969 年に発明された。その原理は、行列 A, B, C を n/2 × n/2 の小行列に分け（図 4）以下の計算を実行することである。

```
1 | M0 = (A00 + A11) (B00 + B11); C00 += M0; C11 += M0;
2 | M1 = (A10 + A11)  B00;        C10 += M1; C11 −= M1;
3 | M2 =        A00  (B01 − B11); C01 += M2; C11 += M2;
4 | M3 =        A11  (B10 − B00); C00 += M3; C10 += M3;
5 | M4 = (A00 + A01)  B11;        C01 += M4; C00 −= M4;
6 | M5 = (A10 − A00) (B00 + B01); C11 += M5;
7 | M6 = (A01 − A11) (B10 + B11); C00 += M6;
```

図 5　Strassen のアルゴリズム

ポイントは n × n の行列積を実行するのに、n/2 × n/2 の行列積が 7 回 (M0 〜 M6 の計算それぞれ一回ずつ）ですんでいることで、この分割を再帰的に実行すると、Θ (nlog 7) の計算量で、n × n の行列積が実行できる。通常のアルゴリズムでは以下の通り n/2 × n/2 の行列積が 8 回必要になり、

```
1 | for (i = 0; i < 2; i++)
2 |   for (j = 0; j < 2; j++)
3 |     for (k = 0; k < 2; k++)
4 |       Cij += Aik * Bkj;
```

Θ (nlog) = Θ (n3) の計算量となる。

Strassen のアルゴリズムは行列積の計算のオーダーを下げることができる、驚くべきアルゴリズムだが、高性能に実装するには問題がある。まず小さい行列に対しての演算数を考えると､掛け算は減らすことができるが足し算･引き算は増えている｡通常のアルゴリズムは、必要な掛け算と足し算の数が一致している。実際のマシンのピーク性能は掛け算と足し算それぞれに限界がある上、fmadd など足し算と掛け算を一度に行う命令を用いて初めてピーク性能を発揮することが多いため、小さい行列に対して Strassen を用いると、足し算・引き算の多さ（掛け算との不均衡）が問題となる。

この問題は通常、ある程度大きな行列に対しては Strassen を適用し、あるところで通常の行列積に切り替えることで回避される。

大きな行列に対しても問題がある。それは、再帰的に掛け算される小行列（図 5 の (A00 + A11), (B00 + B11) など）や、部分解（M0, M1 など）のために、一時領域およびそこへのコピーが必要になることである。一方、通常のアルゴリズムでは、分割後の小行列はもともとの行列 (A, B) の一部に過ぎず、計算結果ももともとの行列 C に直接蓄積すればよかった。大きな行列に対してこのコピーを別途行うと、大量のキャッシュミスが発生する。

従って、Strassen が、その理論的な性質から期待される通りの性能を発揮できるのは、非常に大きな行列に対してのみ、というのがこれまでの conventional wisdom であった（本論文によると、x86 上では n ≧ 2000 程度）。本論文はそれを克服し、n ≈ 500 程度の行列でも、通常の最適化された行列積を上回ることに成功している。

基本的なアイデアはコロンブスの卵的なもので、最適化された行列積について知識があれば、素直に納得できる。最適化された行列積では、TLB ミスを減らす、ハードウェアによる先読みを発動させるため、小行列を適宜連続領域にコピーしてから最内ループを実行する[3]。本論文ではその連続領域へのコピーの一環として、上記で問題となる操作を実行する。具合的には、

M0 = (A00 + A11) (B00 + B11); C00 += M0; C11 += M0;

のような、(行列 + 行列)(行列 + 行列) を実行するのに必要な足し算を、連続領域へのコピーの一環として（小行列ごとに）実行する。また、掛け算を実行した結果も一時領域（上記の M0）に書き込む代わりに、小行列積を実行しながら複数の左辺の行列（上記の C00 および C11）に蓄積する。

結果として、n ≈ 500 程度の大きさの行列で、通常の行列積の実装 (BLIS) を上回ることが確認された。

High Performance Emulation of Quantum Circuits by Thomas Häner, Damian S. Steiger, Mikhail Smelyanskiy, and Matthias Troyer

ETH Zurich, Intel, Microsoft の著者による，High Performance Emulation of Quantum Circuits と題する論文。

量子計算機を古典的な計算機で高速に emulate する方式に関する研究。量子計算機を古典的な計算機で代替させることは、量子計算機が実現するまでの間、プログラムを開発・デバッグする環境として、および、開発途上の量子計算機に対する正しい性能比較対象として、重要である。古典的な計算機で量子計算機を模擬することはこれまでも行われてきたが、この論文では、従来行われきた、``simulation'' に対して、古典的計算機でより効率的に実行できる ``emulation'' を提案する。両者の違いについて説明するため、まず量子計算機の計算モデルを説明する。

n 個の量子ビットからなる系の状態は、それら n 量子 bit が取りうる 2n 個の状態の重ね合わせであり、2n 要素を持つ複素数のベクトル（但しそのノルムは 1）としてモデル化できる。それを状態ベクトルという。量子計算機が 1 ステップで行う計算は、状態ベクトルのノ

3　コピーしない場合，行列要素に対するアクセスが大きなストライドでのアクセスとなりハードウェア先読みが発動されない．また，小行列が TLB の届く範囲でカバーできなくなり，TLB ミスが発生する，行列サイズによってはキャッシュのコンフリクトミスが起きる可能性もある．

ルムを 1 に保つ変換（ユニタリ変換）であり、量子アルゴリズムは、原始的なユニタリ変換を繰り返すことで記述される。原始的なユニタリ変換は、少数（1 個か 2 個）の量子ビットに対して作用する、量子ゲートと呼ばれる変換である。ただしこの場合も状態ベクトルに対する大域的な作用となる。任意の 1 量子ビットに対する量子ゲートと、CNOT と呼ばれる特別な 2 量子ビットに対する量子ゲートで、n 量子ビットに対する任意のユニタリ変換が構成できることが示されている。この事実は、量子計算機を実現する際、CNOT と、1 量子ビットに対する量子ゲートが実現できれば良いという目標を与えてくれる。

量子計算機の simulation とは、アルゴリズムを文字通り、量子ゲートの組み合わせとして模擬するものである。古典的な計算におけるアナロジーとしては、任意の組み合わせ回路が NAND ゲートの組み合わせで実現できるので、与えられた論理式の値を計算するのに、一旦 NAND ゲートの組み合わせとして書き下し、あとは NAND ゲートの作用を模擬しているのと同じである。別の例え話をするならば、スクリプト言語の実行をするのに、スクリプトを一旦機械語の列に変換してから、あとはその機械語を一命令ずつ模擬実行していることに似ている。

これに対して量子回路の emulation とは、実現したいユニタリ変換を、そのまま状態ベクトルに作用させる計算方法である。スクリプト言語のたとえを続けるならば、スクリプト言語に行列計算が出てきたら、その計算自身をまるごと専用の関数呼び出しで実現するのと似ている。Emulation が有用な場面として、多くのアルゴリズムで使われている、量子 FFT や量子位相推定がある。量子 FFT はユニタリ変換であり、その結果を得るには、状態ベクトルに対して文字通り、FFT を実行すれば良い。これが emulation に相当する。一方 simulation は、量子 FFT を結果的に実現する量子ゲートの列（約 n2/2 個の列）を、状態ベクトルに順々に作用させることで、FFT を実現する。前者の方が高速であることは直感的に頷けるであろう。

実験では、状態ベクトルに関する掛け算や割り算などの操作では、数百倍の高速化、量子 FFT でも 6 〜 16 倍の高速化が得られることが確認された。

アイデア自身は、原始的な操作（量子ゲート）を多数組みわせて実現された「高水準な」操作（例えば量子 FFT）を、ゲートまで分解せずにそのまま実行すれば、演算量もベクトルを更新するためのトラフィックも削減できるという、言われてみれば至極当然のものである。むしろこれを今までやってこなかったのが不思議と言える。だが、量子計算機が実用化するまで、その比較対象である古典的なアルゴリズムや、古典的計算機での量子計算エミュレーションを、十分高速化して、正しい比較対象を設定しておくことは重要であろうと思われる。

Daino: A High-level Framework for Parallel and Efficient AMR on GPUs by Mohamed Wahib, Naoya Maruyama and Takayuki Aoki

理研と東工大の著者による、``Daino: A High-level Framework for Parallel and Efficient AMR on GPUs" と題する論文で、最優秀論文賞を受賞した。

AMR は adaptive mesh refinement の略で、偏微分方程式の求解において、細かいメッシュが必要なところだけを細かいメッシュで、そうでないところは荒いメッシュのまま計算をする方法である。流体であれば速度場の変化が激しいところが前者に該当する。

AMR は計算量とメモリ使用量を、低く保ったまま詳細なシミュレーションを可能とする

重要な計算方法であるが、メモリ管理、負荷分散など効率的に実装するのが困難である。そこで本論文は、プログラマが一様なメッシュを仮定した単純なコード（例 : 図 6）に、アノテーション (pragma) を加えるだけで､残りの処理を行ってくれる高水準言語 Daino を設計、実装した。

```
#pragma dno kernel
void func(float ***a, float ***b, ..) {
#pragma dno data domName(i, j, k)
  a, b;
#pragma dno timeloop
  for(int t; t< TIME_MAX;t++) {
    for(int i; i<NX; i++) {
      for(int j; i<NY; j++) {
        ... // comput. not related to a and b
        for(int k; k<NZ; k++) {
          a[i][j][k] = c * (b[i-1][j][k] + b[i+1][j][k] + b[i][j][k]+
                            b[i][j+1][k] + b[i][j-1][k]);
        }
      }
    }
  }
}
```

図 6　Diago のコード例

フェーズフィールド法、流体、shallow-water の 3 つのアプリケーションで、一様メッシュと比べて 1.9 倍から 9.6 倍の高速化を達成するとともに、手動で AMR をコーディングした場合と同等の性能を達成している。フェーズフィールド法の一様メッシュコードは、2011 年にゴードン・ベル賞を受賞したコードである。

Merge-based Parallel Sparse Matrix-Vector Multiplication by Duane Merrill and Michael Garland

NVIDIA の著者による、``Merge-based Parallel Sparse Matrix-Vector Multiplication" と題する論文。疎行列 - ベクトル積の負荷分散を達成する論文。疎行列 - ベクトル積（SpMV; y = Ax; x は m- ベクタ、y は n- ベクタで、A は疎行列）は、

```
for (i = 0; i < n; i++)
  y[i] = 0.0;
  for each の非零要素 A A_{i,j}
    y[i] += A_{i,j} * x[j];
```

図 7　SpMV 擬似コード

で達成できる単純な操作である。ただしもちろん Ai,j を 2 次元の配列として表すわけではなく、最もよく使われるのは、Compressed Sparse Row (CSR) と呼ばれる形式である。これは、A の非零要素だけを行優先でならべた一次元の配列 values と、各行の要素が配列 values においてどこから始まるかを保持する補助配列 row_offsets で表す。 values[row_

offsets[i]], values[row_offsets[i]+1], ldots , values[row_offsets[i+1]]} が i 行目の非零要素となる。さらにそれらの各要素が何列目の要素であるかを示す補助配列 colum_indices も必要になる。それらを用いて、CSR 形式に対する SpMV は以下で実現できる。

```
1 for (int i = 0; i < n; i++) {
2   y[i] = 0.0;
3   for (int nz = row_offsets[i]; nz < row_offsets[i+1]; nz++) {
4     int j = colum_indices[nz];
5     double a = values[nz]; // a = A_{i,j}
6     y[i] += a * x[j];
7   }
8 }
```

図 8　SpMV CSR 形式

課題はこれをどう負荷分散させるかである。上記の CSR を元にしたコードは、行を単位とした負荷分散が自然であり、実際 Intel Math Kernel Library などでも用いられている、しかし一部の行が他と比べて圧倒的に多くの非零要素数を持つ行列に対しては良い負荷分散が得られない。この論文ではそれを裏付けるデータとして、University of Florida の行列コレクション全てに対して得られる実行性能を可視化し、同じ行列サイズでも実行性能が一桁以上違うようなケースがあることを示し、問題を裏付けている。

この問題を解決する一つの方法は、非零要素を等分することで、最も単純には Coordinate (COO) 形式 --- 行列の各要素 Ai,j に対して、行 (i)，列 (i)，値 (Ai,j) の 3 つ組を持つ --- で行列を格納すれば、容易に非零要素を等分割することができる。この方式ではベクトルの同じ要素を複数のスレッドが更新しうるが、各スレッドが担当する最初の行だけは別の変数を用意してそこを更新して、最後に和を取れば競合状態も回避できる。

COO 形式は CSR に比べて領域の利用効率も悪い上、同じ要素に対する書き込みをレジスタ上でまとめることも難しい。そこで、CSR 形式のまま非零要素を等分割することが求められる。まず非零要素を等分割すると、各スレッドが values 配列の中で担当する範囲が決まる。あとはそれに対応する行の範囲を求めればよく、それは row_offsets 配列に対して二分探索を実行することで求まる。

長い前置きのあと、ようやく本論文の新規な部分の説明に入る。本論文が問題としているのは、上記の方法で非零要素を等分割しても、各スレッドが担当する行数に偏りが生ずることがありうる、という点である。これは現実の行列で頻繁に生ずる問題ではないが、非零要素が一つもない行が多数ある場合などに生じうる。Web ページのグラフ構造などでが、このようなケースに該当するらしい。

そこで本論文では、各スレッドが担当する「行の数 + 非零要素数」を等分する。この目標さえ定まれば残りはさほど難しくない。やはり二分探索を使って、各スレッドが担当する非零要素の範囲を求めれば良い。具体的には、スレッド t が担当する最初の非零要素が j 番目の非零要素で、それが i 行目の要素だったとする。するとスレッド 0, …, t - 1 までで合わせて担当する行の数 + 非零要素の数は i + j で、これが割合にして全体の t / T になるように調節すればよい。すなわち、

$$i + j = \frac{(n + nnz)t}{T}$$ 。 (式 1)

ここで j は、それが i 行目の要素であることから、

$$\mathrm{row\_offsets}[i] \leq j < \mathrm{row\_offsets}[i+1]$$

を満たす。すなわち、

$$i + \mathrm{row\_offsets}[i] \leq \frac{(n + nnz)t}{T} < i + \mathrm{row\_offsets}[i+1]$$

となるような i を、やはり二分探索で求め、あとは式 1 に従って j を求めればよい。

実験結果として、行単位での負荷分散を行う Intel MKL や cuSPARSE library と比較して最大で 15 倍程度（調和平均で 20% 程度）の高速化が得られた。行間の負荷分散を解決した他のアプローチと比較すると、キャッシュに収まる小さな行列に対しては大幅な高速化（調和平均で約 11 倍）が得られている。大きな行列に対してはほぼ同等の性能である。

おわりに

次回、SC17 は 2017 年 11 月 12 日から 17 日の日程でコロラド州デンバーにて開催される予定である。

（スーパーコンピューティング研究部門　大島 聡史、田浦 健次朗、中島 研吾）