Vol.2017-HPC-158 No.6
2017/3/8

CPU温度の差異による性能差を考慮したHPL実装

三輪 真弘[1,a)]　福本 尚人[1]　中島 耕太[1]

概要：CPU の冷却条件の違いやチップ毎の消費電力ばらつきにより，CPU 温度がチップ毎に異なり，そのため CPU 動作可能周波数が計算ノード毎に異なる状況の発生が考えられる．このような環境下で並列実行する場合，最も性能の低い計算ノードにプログラム実行が律速されることがある．これを避けるには，各計算ノードの性能差を考慮して，各ノードの処理量を分配するなどの対策が必要である．本研究は，CPU 温度の差異による計算ノード間の性能差を考慮し，システム性能の向上が可能かを明らかにすることを目的とする．実験では，2 ノード間の CPU 周波数差が 20%程度以内ある環境を用いた．ベンチマークとして CPU インテンシブな HPL に対して，ノード毎の処理量を可変とする実装を行い，処理量変更による性能向上が可能か実験を行った．実験の結果，単純な実装では CPU 動作周波数差が 11%程度以内と小さい場合において性能向上が確認できなかったが，不要なメモリコピーを削減することで，CPU 動作周波数差が小さい場合にも処理量変更による性能向上を確認できた．

HPL implementation awaring performance difference caused by the difference of CPU temprature

Miwa Masahiro[1,a)]　Fukumoto Naoto[1]　Nakashima Kohta[1]

Abstract: Difference of power consumptions depending on CPU chips and difference of cooling conditions creates the situations that CPU tempratures differ, and CPU frequency differs. Under this conditions, performance of paarllel executions will be limited by the lowest performant node. To avoid this, some steps like distributing loads with awaring performance difference of each node is required. Our issues in this paper is to make it clear whether we could gain higher performance considering the difference of compute nodes. In our experiment, we used two nodes with the difference of CPU frequency (under 20%). We used CPU-intensive HPL benchmark and implemented the per-node load-changable mechanism to study whether we could achieve higher-performance awaring performance difference. Our experimental results using the naiive implementation show that we could not confirm the performance improvment when the difference of CPU frequency is small (under 11% difference). However we could confirm the performance improvment by reducing the overhead of memory copy in the case where the difference of CPU frequency is small.

1. はじめに

近年の HPC システムで用いられるプロセッサには，CPU 温度に応じて動作可能な周波数が変わる仕組みが導入されている．例えば，Turbo Boost[1] は CPU の消費電力や温度，電流に余裕があれば高い CPU 周波数で動作する仕組みである．また Intel プロセッサにおける AVX 周波数 [2] は，消費電力が高くなる SIMD 演算器を利用している場合に動作クロックが低下するものであるが，これも CPU の熱が許容内であれば動作可能な周波数が向上することがある．

CPU の温度差によって性能差が生じるケースとしてまず冷却条件の違いが考えられる．冷風が適切に届かない場合や，他装置の排熱の影響を受ける場合は，計算ノードの性能が下がってしまう．この対策としてチップの消費電力のばらつきに応じて冷却を調整する方法が考えられるが，装置の複雑化やコストの増加に繋がる．他には，CPU 消費電力のばらつきがある [3]．同じアーキテクチャで同じコア数・周波数を持つ CPU においても，製造ばらつきによっ

[1] (株) 富士通研究所
Fujitsu Laboratories Ltd., Kawasaki, Kanagawa 211–8588, Japan
[a)] masahiro.miwa@jp.fujitsu.com

てチップ毎に CPU 消費電力が異なる場合がある．消費された電力は熱となるため，CPU 電力のばらつきも CPU 温度の違いを生む要因となる．したがって，システム内に異なる CPU 温度の計算ノードが存在することにより，ノード間 CPU 動作周波数差，すなわちノード間性能差が発生する可能性がある．

冷却を個別に調整する方法は装置の複雑化，コストの増加に繋がってしまうため，上述のシステム環境において，異なる CPU 動作周波数差を吸収し，高い性能を達成することは大規模システムでの高性能化という観点で重要である．このための一歩として，CPU 周波数の影響を受けやすい CPU インテンシブなアプリケーションで，性能差を活かすことが可能かどうかを明らかにすることを，本研究の目的としている．

CPU インテンシブなベンチマークである High Performance Linpack (HPL) [10] について計算ノード毎の性能差を考慮した実装を行った．先行研究 [5] の方式と同様の考え方に基づく処理量変更方式を実装し，評価した．実験は，2 台の InfiniBand 接続された IA サーバ（Xeon）を用い，CPU 温度差による CPU 周波数の差異がある環境を想定し，CPU の持つ Dynamic Voltage and Frequency Scaling (DVFS) 機能により，2 台のサーバそれぞれ異なる CPU 周波数に設定した．単純な実装では，大きな CPU 動作周波数差がある場合においては効果が確認できたものの，CPU 動作周波数差が 11%程度以下と小さくなった場合にオーバーヘッド増加により性能向上が確認できなかった．そこで，オーバーヘッド要因の検討を行い，不要なメモリコピーを削減する最適化を行ったところ，CPU 動作周波数差が 11%程度以下の場合にも性能向上が確認できた．

2. High Performance Linpack(HPL)

処理の流れを中心に HPL について説明する．HPL は密行列の連立一次方程式 (Ax=b (A は行列，x, b はベクトルであり，A と b が与えられている) を解くベンチマークであり，A を LU 分解した後，後退代入して解く．スーパーコンピュータシステムの性能ランキングである TOP500[4] では HPL の性能によりランキングされる．本章では，本研究で必要となる部分について説明する．HPL の詳細については [10] を参照されたい．HPL は以下の手順によって処理が進む．

(1) 行列の生成
(2) LU 分解
(3) 後退代入
(4) 結果検証

HPL は Message Passing Interface (MPI)[9] によりプロセス並列化されている．複数プロセス時は，(1) 行列の生成において，行列 A をプロセス間で分割し，各プロセスの担当する行列について生成する．プロセス間のデータ分割をするために，プロセスグリッドを導入している．各プロセスはプロセスグリッド上の 1 ブロックを担当する（以降，プロセスグリッド上のブロックのことを格子ブロックと呼ぶ）．例えば，4 プロセスで並列化する場合，(P, Q)=(2, 2) として設定することができる．ここで P は行方向のプロセス数であり，Q は列方向のプロセス数である．そして，図 2 のように，プロセスグリッドの単位でサイクリックにブロックを割り付けていく．このように並列化時は，各プロセスは行列の一部分を保持する．

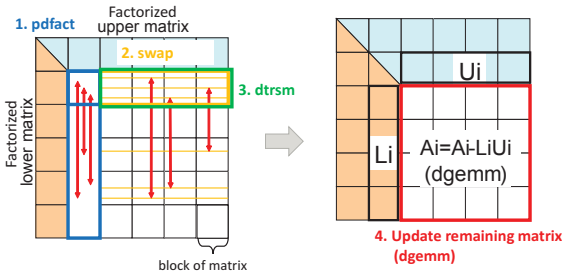

図 1 HPL の処理
Fig. 1 procedure of HPL

行列生成後，(2) LU 分解が呼ばれる．LU 分解が HPL で最も時間の要する処理であり，内部では以下の処理が繰り返し行われる．

LU 分解の処理 (図 1)：
(1) 列パネル分解 (pdfact)
(2) 残行列行入替え (swap)
(3) 行パネル更新 (dtrsm)
(4) 残行列更新 (dgemm)

LU 分解の処理はブロックサイズ単位のフェーズで繰り返すことで計算が進む．残行列のうち最も左の列をピボット選択しながら LU 分解する（列パネル分解）．次に残った行列に対してピボット選択を反映する（swap）．その後，行パネルを更新し（dtrsm），最後に残った行列に対して行列積を行う（dgemm）．1 フェーズ進むと次のブロックに進み，残行列が 1 プロセスずつ減っていき，残行列がなくなると LU 分解が完了する．上述した通り，行列データは各プロセス間に分割されているため，処理上で他プロセスが保持するデータは通信によって交換される．また，HPL 実行パラメータの Depth 値によるが，列パネル分解後の列パネルデータの転送において，アップデート処理（swap, dtrsm, dgemm）実行中にオーバーラップさせる実装が行われている．これにより転送の待ち時間を隠蔽することによる性能向上が期待できる．通信処理を進めるために，HPL_Bcast 関数を呼び出すが，呼び出しは NB サイズ毎

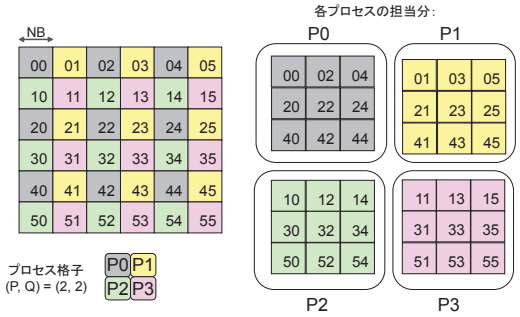

図 2 HPL プロセスグリッド
Fig. 2 HPL process grid

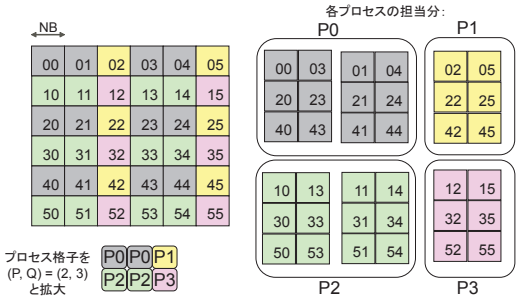

図 3 処理量変更方式
Fig. 3 the way of changing loads of HPL

のブロック単位で行う dgemm の間に呼ぶよう実装されている．

3. HPL 処理量変更方式

処理量変更を可能とする HPL 実装方式について述べる．処理量を変更する方式として 2 つの方法が考えられる．1 つはサーバ毎に割り当てる HPL プロセスの数を変える方法であり，もう 1 つの方法はプロセス数は同じにしてプロセス毎に処理する行列サイズを変更する方式である．先行研究 [5] に詳しいが，前者の方式はプロセス数増加により，プロセス間通信が増加してしまうことがある．また，ノード内のコア数によってプロセス数の比率は制限されてしまう．例えばノード a がノード b の 1.33 倍の性能となった場合を考える．この場合，ノード a にはノード b の 1.33 倍のプロセス数を割り当てたい．ノード a, b のコア数が 12 だとすると，ノード a は 3 コアずつの 4 プロセスで実行し，ノード b は 4 コアずつの 3 プロセスで実行できる．しかし，ノードのコア数が 8 コアの場合は，プロセス数を 4:3 で割り当てようとするとコアの余りが生じてしまい，性能低下する．特にノード間の性能差が小さい場合，コアの余りが生じやすい．このようにプロセス数の比率を変える方法では，性能差が小さい場合の対応が難しい．

そこで CPU 動作周波数差がそれほど大きくない場合に処理量を変更する方式として，プロセスが担当する格子ブロック数をプロセス毎に変更する方式を本研究で採用している．このアイデアは先行研究 [5] で発表されている．

この方式は処理量変更を実現する上で実装が容易というメリットがある．それは，HPL の処理の多くは，自身の担当する格子ブロックの座標に基づき処理しているためである．たとえば HPL_numroc という HPL 内部で多用されている関数があるが，格子ブロックの座標を与えるとその格子ブロックが担当する行列の行数や列数を返す．したがって，処理量を変更する方法として格子ブロック数を調節する方法は，これらの HPL ユーティリティ関数の実装は変更せずそのまま利用できる．

処理量変更方式を実現するために，プロセスグリッドの

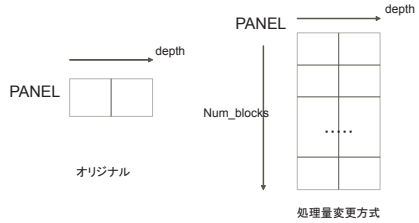

図 4 処理量変更方式における PANEL
Fig. 4 PANEL data structure in ways of changing loads

拡大を行う．図 3 では，プロセス 0 (P0) とプロセス 2 (P2) の処理量を，プロセス 1 (P1) ，プロセス 3 (P3) に対して 2 倍の処理量を与える場合を示している．2 倍の処理量を与えるプロセスは 2 倍の格子ブロックを割り当てることで，処理量変更が実現できる．処理量変更方式では，各プロセスは複数の格子ブロックを担当することがあるため，HPL 処理で主に用いられる PANEL データ構造を変更している．オリジナル版では 1 つの格子ブロックを担当しているが，処理量変更方式では各プロセスが複数のブロック (Num_blocks) を担当する．複数の格子ブロックを担当するプロセスは，それぞれの格子ブロックに対し，列パネル分解や update (swap, dtrsm, dgemm) を実施する．

本実装では，列方向に対してのみ格子ブロック数を変更する処理量変更を行っている． 各フェーズでは，分解済みの列パネルをブロードキャストするが，この際の通信は，元のプロセスグリッドに基づき行う．各フェーズにおいて列パネル分解の対象となるのはプロセスグリッド上のある 1 列にある格子ブロックであり，このブロックを保持するプロセスは，列パネル分解を行い，このパネルを送信する．列パネル分解の対象となるブロックを保持していないプロセスは受信を行う．処理量変更方式では，同プロセスにおける他の格子ブロックについて Bcast された列パネルデータを反映する必要があるが，元のグリットに基づく送受信後には，プロセス内で 1 つの格子ブロックについて Bcast された列パネルを持つ．そこで同一プロセス内の他の格子ブロックに関する PANEL からこの列パネルを参照すればよい．行方向について元のグリッドと変更がないため，縦方向に関する swap 処理について変更なく実行できる．

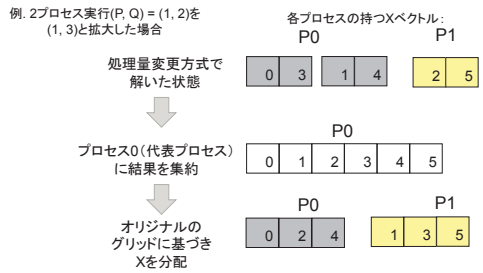

図 5 処理量変更方式における結果検証のための X ベクトルの分配
Fig. 5 Distribution of X vector for validation in the way of changing loads

次に結果検証部分の修正について述べる．LU 分解後に後退代入を行うことで，解 X ベクトルが得られる．この解の検証では，行列 A を再生成してチェックのための関数群が実行される．検証を行う場合，拡大したプロセスグリッドのまま行う方法も考えられるが，本研究ではオリジナルの処理をそのまま活用するため，解の検証は元のグリッドのまま行っている．そのため，プロセスグリッドの拡大を行った後に得られた X ベクトルを，図 5 に示すように元のグリッドに対応するよう分配している．

4. 実験

HPL 処理量変更方式の効果を検証するための実験を行う．

Xeon 2 台（サーバ A, サーバ B）の環境で，2 サーバ間は InfiniBand (FDR) で接続されている．各サーバは，CPU E3-1240 v3 (定格 3.4GHz, 4 コア) を 1 つずつ有しており，メモリは DDR3 で 16 GB 搭載されている．OS は Red Hat Enterprise Linux Server release 6.4 を用いた．HPL は HPL-2.1 を利用しており，オリジナル版はこれをそのまま実行している．処理量変更方式は，HPL-2.1 に実装した．BLAS は Intel MKL (2016.4.258) を用い，MPI は，Intel MPI 5.0.2 を用いた．

実験のために 2 台のサーバの CPU 周波数を以下の 3 パターンに設定し，性能の測定を行った．
（サーバ A CPU 周波数，サーバ B CPU 周波数）＝
(1) (3.4 GHz, 2.8 GHz) (CPU 動作周波数差 17.6%)
(2) (3.4 GHz, 3.0 GHz) (CPU 動作周波数差 11.7%)
(3) (3.4 GHz, 3.2 GHz) (CPU 動作周波数差 5.8%)

実行プロセス数は，ノード辺り 2 プロセスの計 4 プロセスで実行した．実行のプロセスグリッドは，(P, Q) = (1, 4) とした．処理量変更方式での処理比率として，[5] を基に，CPU 周波数の比とした (以下)．その他実験に用いた HPL の実行パラメータを表 1 に示す．測定は 3 回測定した値の平均を採用している．

- (CPU 動作周波数差 17.6%) = (17 : 14)
- (CPU 動作周波数差 11.7%) = (17 : 15)

表 1 実験に用いた HPL 実行パラメータ
Table 1 HPL parameters used for experiments

	value
Ns	40000
NB	168
threshold	16.0
PFACTs	Right
NBMINs	2
NDIVs	2
RFACTs	Right
BCASTs	1 ring
DEAPTHs	1
SWAP	long
swapping threshold	192
L1 in form	no-transposed
U in form	no-transposed
memory alignment in double	8

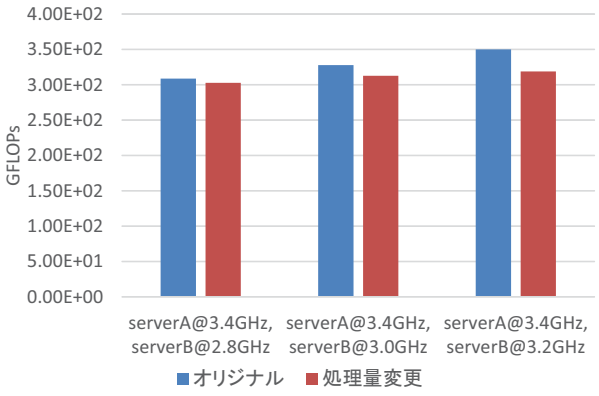

図 6 実験結果
Fig. 6 Experimental result

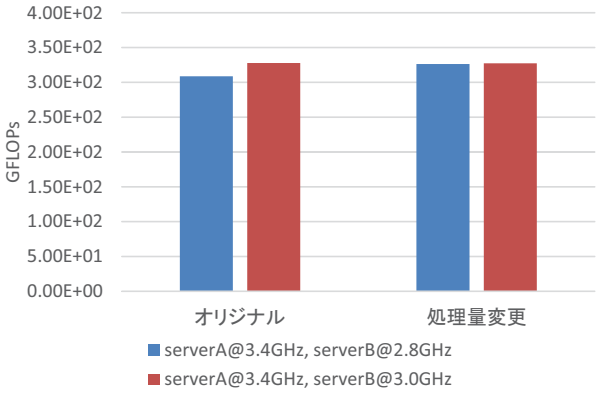

図 7 実験結果 (処理比率の変更)
Fig. 7 Experimental result(changing the number of load ratio)

- (CPU 動作周波数差 5.8%) = (17 : 16)

4.1 結果

測定結果を図 6 に示す．横軸は実験の各パターンであ

り，縦軸は HPL が出力する性能値である．この測定では，処理量変更方式による性能向上が得られていない．この測定では，担当ブロック数が多くなっており，この影響でないかを確認するため，CPU 動作周波数差そのままの比率ではないが，担当ブロック数が少なくなる以下の比率で再度測定を行った．ただし，5.8%の動作周波数差に近い担当ブロック数が少なくなる比率は見当たらないため除いている．その測定結果を図 7 に示す．

- (CPU 動作周波数差 17.6%) のとき (6 : 5) (16.6%)
- (CPU 動作周波数差 11.7%) のとき (9 : 8) (11.1%)

CPU 動作周波数差が 17.6%の場合では，オリジナル版に対し，処理量変更方式による性能向上が確認できた．一方，CPU 動作周波数差が 11.7%の場合では，効果が見られない．

5. ノード間 CPU 動作周波数差が小さい場合の改善

CPU 動作周波数差が 11.7%よりも小さい場合において処理量変更による効果が見られなかった原因について調査する．これらのパターンでは CPU 動作周波数差が小さくなり，それに応じた処理比率を実現するために格子ブロック数を増加する必要がある． 格子ブロックが増加した場合にオリジナル版と異なる挙動をする主な点として，処理量変更方式では，グリッドの拡大を行うために，1 格子ブロック辺りの担当する行列サイズが小さくなる．一方，1 プロセスで複数の担当ブロックを処理するため，dgemm 呼び出し回数が増加する．文献 [8] によると，Intel MKL の行列積では，最適な性能を出すために，オリジナルの入力行列を各ターゲットプラットフォームに合わせた内部データフォーマットで保持することがわかった．このため，dgemm を行う際の列パネル L が複数回呼び出す dgemm によって，何度も内部データ形式へのコピーが発生する．この列パネルは同じものであるため，一度のバッファへのコピーで済ますことができれば，オーバーヘッドの削減ができる．文献 [8] によると，Intel は Intel Packed API を提供しており，これにより dgemm 演算時のコピーを削減することができる．通常の dgemm では内部バッファへのコピーと演算が 1 つになっていたが，packed API を使うことで内部バッファのコピーと演算を分けることができる．

Intel Packed API:

- cblas_dgemm_alloc (メモリ割り当て)
- cblas_dgemm_pack (メモリコピー)
- cblas_dgemm_compute (演算)
- cblas_dgemm_free (メモリ開放)

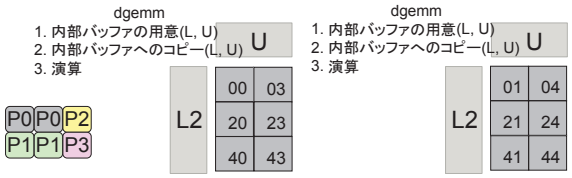

図 8 通常の dgemm
Fig. 8 normal dgemm

図 8 と図 9 に通常の dgemm と Intel Packed API を用いた場合についてそれぞれ示す．例として，図 2 における P0 に着目して説明する．図 8 では，列パネル L のコピーが 2 回発生している．CPU 動作周波数差が小さい場合に対応する処理量を実現するためにこのコピーが増えてしまう．例えば 2 倍の CPU 動作周波数差であれば，2:1 と処理量を変更すればよいが，10%の CPU 動作周波数差の場合，10:9 といった割り振りになるために，CPU 周波数が大きい計算ノードは 10 個の格子ブロックすなわち 10 回のコピー，CPU 周波数が小さい計算ノードは同様に 9 回のコピーとなる．図 9 では，列パネル L のコピーは一度で済んでいることがわかる．

Intel Packed API を用いた処理量変更方式の結果を図 10 に示す．Intel Packed API を用いた版は，通常の dgemm を用いた場合より改善され，オリジナル版を超える性能を達成した．図 10 には，「オリジナル packed API」があるが，これは，本論文 2 章でも述べたが，列パネルの転送が完了するまでは，NB の単位で dgemm を呼ぶため，オリジナル版においても不要なコピーが発生する可能性があるため，この影響を調べるために測定した．また，「ideal」はオリジナル版の測定結果を基に，理想的に達成される性能を示している．[5] の測定にあるように，HPL 性能は低いノード性能に制限されるため，オリジナルの測定結果は，低い CPU 動作周波数の計算サーバ 2 台による性能と考えられる．理想的には，高い CPU 動作周波数分の性能向上が期待される．例えば，17.6%の CPU 動作周波数差がある 3.4 GHz と 2.8 GHz のケースでは，(17.6/2) = 8.8%の向上が期待される．測定結果をみると，メモリコピーを削減した処理量変更方式の性能が，実験で用いたパターンのいずれもオリジナルの性能を超えた．3.4 GHz と 3.2 GHz 以外のパターンでは，メモリコピーを削減した処理量変更方式で「ideal」に近い値が得られている．3.4 GHz と 3.2 GHz といった CPU 動作周波数差が小さい場合でも，メモリコピーを削減した処理量変更方式では，オリジナルに対し，1.6%性能向上できた (表 2).

6. 関連研究

HPL の性能向上のために，ノード内アクセラレータを合わせて活用する研究が行われている [6][7]．われわれは同一プロセッサのみを有する計算ノードから構成されるシス

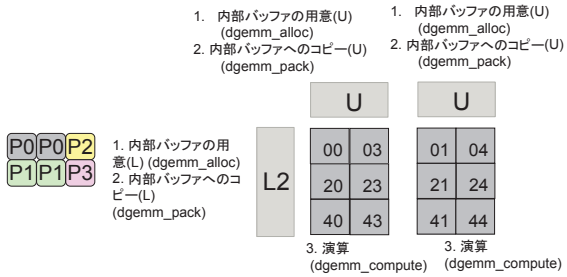

図 9 packed API を用いた dgemm(メモリコピーの削減)
Fig. 9 dgemm using packed API(reduction of memory copy)

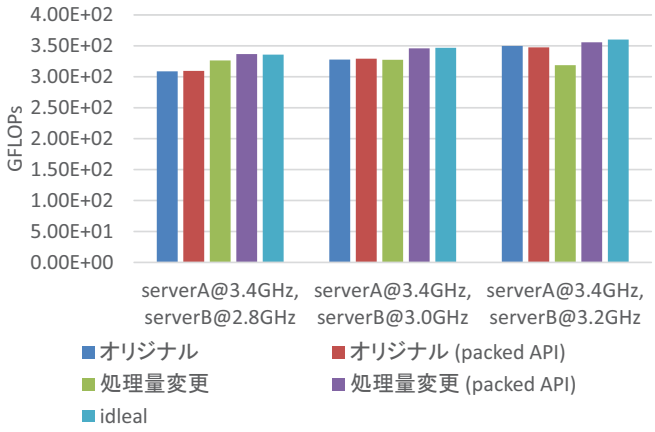

図 10 実験結果 (メモリコピーの削減)
Fig. 10 Experimental result(reduction of memory copy)

表 2 実験結果 (メモリコピーの削減) (3.4 GHz, 3.2 GHz)
Table 2 experimental result (reduction of memory copy) (3.4 GHz, 3.2 GHz)

	GFLOPs	オリジナルからの向上 (%)
オリジナル	3.50E+02	0.0
オリジナル (packed API)	3.47E+02	-0.7
処理量変更	3.19E+02	-8.9
処理量変更 (packed API)	3.56E+02	1.6

テムにおいて，CPU 温度の違いから生じる CPU 動作周波数差 (性能差) がある環境を対象に性能向上を行った．

笹生らは，HPL のプロセス毎の処理量を変更する方式について我々が知る限り最初に発表している [5]．笹生らはコモディティークラスタが持つ不均質性を考え，異なる種類の CPU が混在する環境下で処理量を変更する HPL の評価を行っている．評価は 2 倍の動作周波数差のある CPU を混在した環境で行っている．われわれは CPU 温度の違いから生じる CPU 動作周波数差として 20%程度以内の比較的小さい差を考えており，そのような CPU 動作周波数差が小さい中でシステム全体性能を引き出すために必要なオーバーヘッドの分析をし，性能を引き出すことに成功している．

7. おわりに

CPU 温度により生じる性能差 (CPU 動作周波数差) のある環境向けに，性能差に応じた処理量変更を行うことで HPL の性能を向上することができた．単純な実装では，CPU 動作周波数差が小さい場合には性能向上が確認できなかったが，性能分析を行い，メモリコピーを削減した版では性能向上が確認できた．この取り組みを通して，小さい計算ノード間性能さを含むシステムでは，性能向上のために注意深い最適化が必要となることがわかった．

今後の課題として，大規模実行による効果検証や，CPU 動作周波数の違いの種類がいくつか存在する場合の効果検証がある．また，その他のアプリケーションや実装を対象に効果検証を進め，CPU 温度の差異による CPU 動作周波数差のある環境にも対応するアプリケーションの開発についての研究がある．

参考文献

[1] Intel Turbo Boost Technology 2.0. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html
[2] Intel White Paper ”Optimizing Performance with Intel Advanced Vector Extensions”
[3] Joakim v. Kistowski, Hansfried Block, John Beckett, Cloyce Spradling, Klaus-Dieter Lange, Samuel Kounev, ”Variations in CPU Power Consumption,” ICPE’16, March 12-18, 2016, Delft, Netherlands.
[4] TOP500 Supercomputer Sites, https://www.top500.org/
[5] 笹生 健，松岡 聡，建部 修見，ヘテロなクラスタ環境における並列 LINPACK の最適化，HPC86-9 (2001.5.25).
[6] 遠藤 敏夫，額田 彰，松岡 聡，ヘテロ型スーパーコンピュータ TSUBAME2.0 の Linpack による性能評価，HPC-128 No.11 (2010.12.16).
[7] Cheng Chen, Jianbin Fang, Tao Tang, Canqun Yang, ”LU factorization on heterogeneous systems: an energy-efficient approach towards high performance,” Springer (Published online: 02 January 2017).
[8] Intel: Introducing the new Packed APIs for GEMM, https://software.intel.com/en-us/articles/introducing-the-new-packed-apis-for-gemm
[9] Message Passing Interface(MPI) Forum: http://mpi-forum.org/.
[10] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers: http://www.netlib.org/benchmark/hpl/