

動的なノード数変更に応じてプロセス単位で負荷分散を行う MPI環境の実現

澤田 祐樹^{1,a)} 大津 金光^{1,b)} 大川 猛¹ 横田 隆史¹

概要：近年モバイル端末はマルチコアプロセッサの搭載により高性能化が著しく、並列分散アプリケーションのためのプラットフォームとして新たに注目されている。我々は Android OS を搭載した端末を計算ノードとして活用し、MPI アプリケーションを並列処理するクラスタシステムを開発している。本システムは端末の脱退、参入に伴うノード数の動的変化に対応するためにチェックポイントデータの取得と、チェックポイントデータからの並列処理のリスタートが可能である。しかし本システムでは、従来のリスタート機能においてノード内の複数タスクを不可分に移動することしかできない制限があった。そのため、ノード数が動的に変化した場合、クラスタ内のノード間における並列タスクの負荷に偏りが生じる場合があった。これを解決するために、ノード内の複数タスクをそれぞれ任意のノードに移動することを可能とする。それにより、並列タスクの負荷分散を行う MPI 並列処理環境を実現し、その効果を明らかにする。

1. はじめに

近年、Android OS や iOS に代表されるスマートフォンやタブレット端末といったモバイル端末が急速に普及している。これらのモバイル端末の多くがマルチコアを搭載しており、現在ではオクタコアを搭載した端末も登場し、性能を低下させることなく複数の並列タスクを処理できるため、端末単体の処理性能が大幅に向上している。また、それ以外にも、ローカルストレージ容量の増大やネットワーク性能の向上も大きい。今日このような高性能なモバイル端末を多くの人が携帯している。そのため、モバイル端末は並列分散アプリケーションを並列処理するための新しいプラットフォームとして注目されている [1], [2], [3], [4]。関連研究 [1] では、Wi-Fi や WiMAX といった無線通信で接続し、モバイル端末をグリッドコンピューティングに活用している。このように多くのモバイル端末を並列分散処理の計算ノードとして活用し、サーバーの代用とする事も可能である [2]。また、省電力の観点から Bluetooth を通信手段とし、モバイル端末で並列分散処理をする事例もある [3]。そこで、我々は Android OS を搭載したモバイル端末を計算ノードとし、MPI 並列プログラムを実行するためのクラスタシステム (Android クラスタシステム) を開発している。クラスタ内のそれぞれのモバイル端末は相互接

続し並列分散アプリケーションを並列処理する。

計算ノードであるモバイル端末には移動が発生し、通信途絶によるクラスタからの脱退やクラスタへの新たな端末の参入が想定される。どの端末が利用可能であるかを把握することが必要であるため、本システムでは定期的なメッセージ交換により利用可能な端末群を把握する [5]。また、途中で脱退した場合でも並列処理を継続可能とするためにチェックポイント/リスタート機能を導入しており、これによって並列分散アプリケーションが中断した場合でも、チェックポイントデータからアプリケーションをリスタートすることが可能である [6]。

しかしながら、従来システムではリスタート機能に制約があり、ノード内の複数タスクを不可分に移動することしかできない。つまりノード単位でしか並列タスクを移動することができない。そのため、クラスタ内のノード間における並列タスクの負荷に偏りが生じる場合がある。例えば、ノードの脱退によって並列アプリケーションが中断した場合、クラスタに残った別のノードが脱退したノード (脱退ノード) の全ての並列処理を引き継いでリスタートする。そのため、脱退ノードが担っていた並列処理を引き継いだノードだけ並列タスクの負荷が高くなり、クラスタ全体の性能が低下する。

そこで、クラスタシステム上で効率的な並列処理を行うために、リスタート時に各ノードにおける並列タスクの負荷をノード単位ではなく、プロセス単位で分散する機能を追加した。本稿では開発した負荷分散機能の実装とその効

¹ 宇都宮大学 大学院 工学研究科
Graduate School of Engineering, Utsunomiya University

a) sawada@virgo.is.utsunomiya-u.ac.jp

b) kim@is.utsunomiya-u.ac.jp



図 1 Android 端末を用いたクラスタシステム

果について述べる。

2. Android クラスタシステム

我々が開発している、モバイル端末を計算ノードとしたクラスタシステムについて述べる。

2.1 クラスタシステム環境

クラスタシステムを構成するノードとして、Android OS を搭載したモバイル端末 (Android 端末) を使用する。モバイル端末の OS 別出荷台数は Android OS が最も多いため、クラスタの構成ノードを Android 端末とすることで、クラスタシステム (Android クラスタシステム) が形成しやすいと考える。図 1 に Android クラスタシステムの構成ノードを示す。

本システムでは現在、ノード間の通信手段として WiFi による無線通信を使用している。複数ノードを利用して並列分散処理を行う際、通信性能はクラスタ全体の性能に大きな影響を及ぼすため、高速な無線通信手段を用いる必要がある。そこで、本システムでは多くのモバイル端末で使用でき、かつ高速な通信が可能な Wi-Fi を無線通信手段として利用する。

2.2 並列アプリケーション実行環境

並列分散処理のフレームワークとして、Message Passing Interface (MPI) の実装の 1 つである Open MPI を使用する。Open MPI はオープンソースであり、機能の追加や変更が可能である。

Android OS 上で動作させるアプリケーションは、一般的に Java 言語で記述される。コンパイルされたバイトコードは DalvikVM と呼ばれる仮想マシン上で実行される。仮想マシン上での実行は、CPU が直接ネイティブコードを実行するよりも遅い。そこで、Android NDK (Native Development Kit) [7] を使用し、アプリケーションプログラムを C/C++ で記述する。このプログラムソースをコンパイルすることで、CPU が直接実行するネイティブコードが生成され、仮想マシン上での実行よりも高速な実行が可能となる。したがって、我々は Open MPI とクラスタ上で動作する MPI アプリケーションを Android NDK を使

用してビルドする。これにより、CPU のネイティブ実行可能な MPI による並列分散処理が Android クラスタシステム上で実現できる。

2.3 チェックポイント/リスタート機能

MPI アプリケーションを実行中に、クラスタからノードが脱退した場合、MPI アプリケーションの継続が不可能となる。一般的にノードの脱退を含め、何らかの理由で並列処理が中断した場合、同じ MPI アプリケーションを最初から並列実行し直す必要がある。このオーバーヘッドを抑えるために、我々はチェックポイントング技術を採用する。

MPI アプリケーションの中断は予期できないため、中断する前に並列処理を行う各プロセスの状態を保存しておく必要がある。さらに、クラスタから脱退したノードの並列処理を、クラスタ内の別のノードへ移譲できる必要もある。チェックポイントング技術はこの 2 つの要求を満たす。チェックポイントング技術にはチェックポイントとリスタートの 2 つの機能がある。本システムにおいて、チェックポイント機能は MPI 並列プロセスの状態を記録し、リスタート時にはチェックポイント時に記録したプロセスイメージを使用して MPI アプリケーションをリスタートする。

2.3.1 チェックポイントングソフトウェア

ユーザレベルで並列プロセス群のチェックポイントデータの取得および復帰を可能とするために、我々は DMTC (Distributed MultiThreaded Checkpointing) [8] を使用する。DMTC はユーザレベルでのチェックポイントングをサポートしており、マルチスレッド実行にも対応している。x86 や ARM, MIPS の各命令セットに対応しており、カーネルに非依存である。そして、アプリケーションを変更することなく、チェックポイント/リスタートが可能である。

2.3.2 チェックポイント機能

DMTC によるチェックポイントを行うにはまず、`dmtcp_launch` コマンドによりアプリケーションの実行を開始する必要がある。アプリケーションの開始時には、アプリケーションを実行したノード (ホストノード) 内にプロセスの管理や制御を行うデーモンプロセス (`dmtcp.coordinator`) を立ち上げる。また、DMTC の管理下にある各プロセス内には、チェックポイントスレッド (CT) が起動し、チェックポイント取得要求に応じて、プロセスのチェックポイントを行う。`dmtcp_command` コマンドを用いたチェックポイント取得要求もしくは、直接 DMTC の API 関数 (`dmtcp_checkpoint()`) をコールすることで `dmtcp.coordinator` が各プロセス内の CT にチェックポイント取得を要求する。CT は一時的にプロセス内のユーザスレッドを停止させた後、プロセスのチェックポイントデータ (ckpt) を生成する。その後、CT はユーザス

レッドの停止状態を解き、プロセスは実行を再開する。

Open MPI で並列処理を開始すると、ホストノード (MPI アプリケーションを起動したノード) には `orterun` プロセス、それ以外のノード (リモートノード) では `orted` プロセスが起動する。この2つのデーモンプロセスは MPI 並列プロセスの生成/管理を行う。我々は、`dmtcp-coordinator` に対してチェックポイントの取得要求を行う `dmtcp_checkpoint()` 関数を `orterun` に追加することで、チェックポイント取得を MPI 側でコントロールできるようにした。この時、ノードの脱退を予測することは困難であるため、チェックポイントの取得要求は定期的に行う。そして、脱退ノードの並列タスクを別のノードによる継承を可能とするために、各リモートノードで生成された `ckpt` をホストノードに集約する制御を加える。3 ノードで構成され、各ノードで 2 プロセス実行しているクラスタを例とし、MPI アプリケーション実行中における `ckpt` の移動の様子を図 2 に示す。ノードの脱退が発生した場合は、ホストノードに集めていた脱退ノードの `ckpt` を別のノードへ送信し並列タスクを継承させる。

2.3.3 リスタート機能

DMTCP におけるリスタート機能は `ckpt` を元にプロセスを復元し、並列アプリケーションの実行を再開させる。図 2 のクラスタにおいて、リモートノード B が脱退した場合の、リスタート処理について述べる。図 2 中の P1 から P6 は実行中の MPI 並列プロセスを示している。プロセス間における通信途絶の検知は Open MPI が行っている。MPI プロセスは各 TCP コネクションの通信途絶を検知し、途絶したソケットを閉じるといった制御を行っている。そこで、ホストノード内の MPI 並列プロセスがリモートノードとの通信途絶を検知しソケットを閉じる際、通信途絶を検知した MPI 並列プロセスは `orterun` にシグナルを送信する。図 2 の例ではリモートノード B が脱退し、シグナルを受け取った `orterun` は `dmtcp-coordinator` 経由で、一度全ての MPI 並列プロセスを強制終了させる。そして脱退したリモートノード B で起動していたプロセ

ス (P5, P6, `orted`) を別のリモートノードに移譲する。この例ではリモートノード A が脱退ノードのプロセスを継承する。まずホストノードはリモートノード B の `ckpt` をリモートノード A へ送信する。DMTCP は並列処理を行うノードの指定が可能なホストファイルを持つ。ホストファイルには並列処理を行うノードのホスト名を記述されている。このホストファイルを使用し、ファイル内において脱退ノードの並列タスクを行うホスト名の指定部分を別のリモートノードのホスト名に置き換えることで、脱退ノードのプロセスを別のリモートノードが継承してリスタートすることが可能となる。しかし、この例では脱退ノードのプロセスを継承したリモートノード A で起動するプロセス数がだけ増加する。

例えば、図 2 において、リモートノード B が脱退し、リモートノード B の並列タスクをリモートノード A が継承してリスタートした場合、ホストノードでは 2 つの並列プロセス (P1 と P2) が起動し、リモートノード A では 4 つの並列プロセス (P3, P4, P5, P6) が起動する。リモートノード A だけ並列処理の負荷が高くなり、クラスタ全体の性能が著しく低下する。もし脱退ノードで起動していた 2 プロセスをホストノードとリモートノード A に分散して移譲することが可能ならば、それぞれのノードで起動するプロセス数は同じとなり、効率的な並列処理を行える。

3. プロセス単位での負荷分散

従来の DMTCP のリスタートでは、ノード単位による並列プロセスの割り当てによって処理性能が大きく低下する場合があった。そこで処理性能の低下を緩和するための負荷分散機能を追加する。

3.1 負荷分散の概要

負荷分散はノード単位ではなく、プロセス単位で並列プロセスを割り当てることを可能とする。リスタート時にプロセス単位での並列処理の移譲を行うことで、ノード間における並列タスクの負荷バランスを均等になるように調整し、効率的な並列処理を実現する。そのために必要な事はプロセス単位で並列プロセスを復元可能とすることである。

3.2 負荷分散プロセスの生成

通常の DMTCP ではアプリケーションをリスタートする場合は `ckpt` 作成時に同時に作成されるヘルパースクリプトを使用する。ヘルパースクリプト内では `dmtcp_restart` コマンドを使用して、各ノードごとにどの `ckpt` のプロセスを復元するかを制御する。復元すべきプロセスには順序関係があり、MPI の管理プロセスを復元してから MPI 並列プロセスの復元を行う。リスタート用のヘルパースクリプトを実行してから、MPI 並列プロセスを復元するまでのプロセスの生成/変化シーケンスを図 3 に示す。図 3 はホ

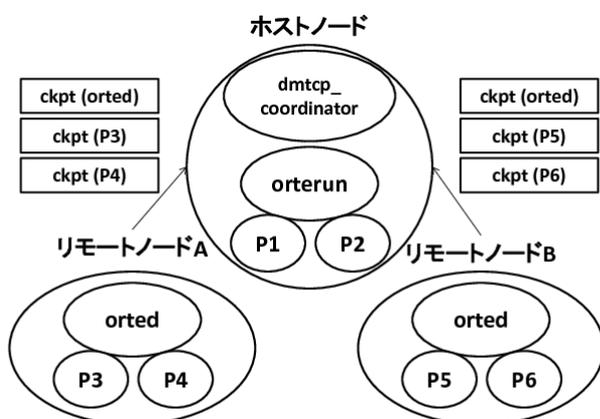


図 2 3 ノード構成のクラスタシステム

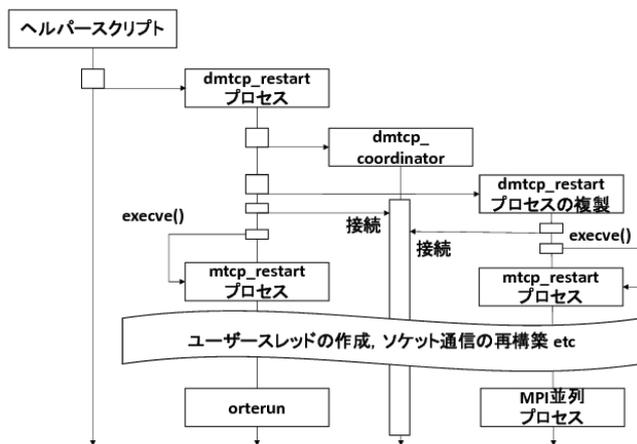


図 3 リスタート時のプロセスの遷移シーケンス

ストノードにおけるプロセスの生成シーケンスを示している。dmtcp_restart コマンドのプロセスは読み込んだ ckpt を参照し、復元する MPI 管理プロセスと親子関係にある ckpt を復元対象とし、fork() システムコールを実行して、dmtcp_restart プロセスを複製する。また dmtcp_restart コマンドのプロセスは dmtcp_coordinator も生成する。リモートノードにおける復元対象のプロセスは、ヘルパースクリプトによって実行される ssh コマンドを経由して生成された dmtcp_restart プロセスがホストノードと同様に親子関係にある ckpt を復元対象とし、dmtcp_restart プロセスを複製する（ただし、リモートノードにおいては dmtcp_coordinator の生成は行わない）。

dmtcp_coordinator とデータをやり取りしながら復元処理を行うため、各プロセスは dmtcp_coordinator と接続する。その後、各プロセスは mtcp_restart プロセスへと変化し、チェックポイント時に開いていたファイルの再オープンやソケット通信を含めた各種ファイルディスクリプタの復元等を行う。

プロセス単位での割り当てを可能にするには、まず移譲させるプロセスの ckpt を移譲先ノードで実行される dmtcp_restart コマンドの引数として与え、fork() が実行される必要がある。しかしチェックポイント時にプロセスの親子関係が記録されるため、親子関係のないプロセスの ckpt は復元対象とされず fork() は実行されない。DMTCP に変更を加え、親子関係のないプロセスの ckpt でも fork() し、dmtcp_restart プロセスが複製されるようにした。

3.3 プロセス間通信の再構築

dmtcp_restart プロセスの複製を生成するだけではリス

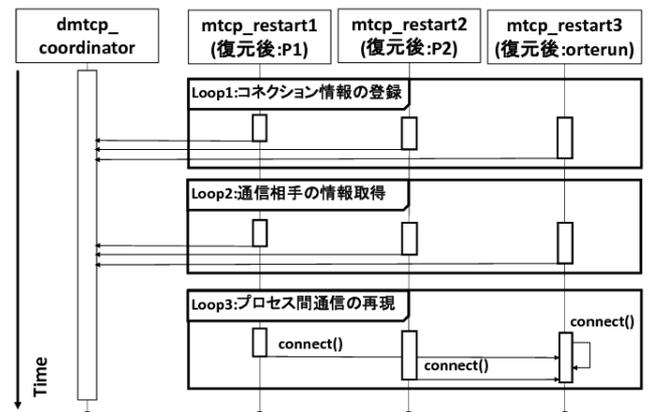


図 4 プロセス間通信の再構築シーケンス

タートすることができない。チェックポイント時にはプロセス間通信の情報も記録される。通信の送信側と受信側でユニークな ID (コネクション ID) を持ち、互に対応関係にある。このコネクション ID とともに、通信に使用したソケットファイルディスクリプタも記録される。リスタート時にはこの対応関係を元にプロセス間通信を再構築する。2 プロセス (P1, P2) 起動しているホストノードを例として、DMTCP がプロセス間通信を再現するまでのシーケンス図を図 4 に示す。プロセス間通信の再構築時、各プロセスは mtcp_restart プロセスの状態である。図 4 の各処理において、mtcp_restart プロセスは状態確認のために dmtcp_coordinator とメッセージ交換を行いながら、データのやり取り/処理を行う。まず mtcp_restart プロセスは、受信のコネクション ID と自身のアドレス情報を dmtcp_coordinator に登録する (Loop1)。自身が記憶するコネクションを全て登録完了後、次に各プロセスは接続先のアドレス情報を取得を行う (Loop2)。接続先のコネクション ID で dmtcp_coordinator へ問い合わせ、接続先のアドレス情報を取得する。そして取得したアドレス情報を用いてプロセス間通信を再構築する (Loop3)。各ノード内の MPI 管理プロセス (orterun, orted) は子プロセスである MPI プロセスとのコネクションを持つため、MPI 並列プロセスは MPI 管理プロセスとのコネクションを復元する (図 4 の Loop3 は再構築するコネクションの一部だけ示している)。

プロセス単位で負荷分散を行うと一部復元できない通信が発生する。図 2 においてリモートノード B が脱退し、クラスタに残った 2 ノードにそれぞれ 1 プロセス (P5, P6) ずつ分散させてプロセスを割り当てる場合を想定し、図 5 に示す。リスタート時、脱退ノードで起動していたプロセス (P5, P6) を復元するためにそれぞれのノードで

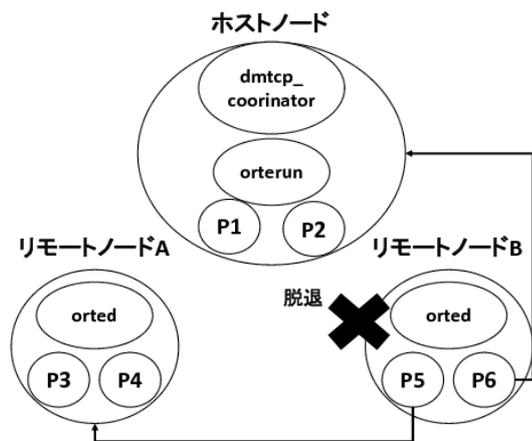


図 5 プロセス単位での割り当て (負荷分散) の動作例

dmtcp_restart プロセスの複製を作成する。mtcp_restart プロセスへ変化後、プロセス間通信の再構築を開始する。しかし脱退ノード内の orted との間の通信を復元できず、リスタートすることができない。

3.3.1 コネクションの変更

そこで別のノードへ移譲した MPI 並列プロセスにおいて、脱退ノード内の orted と通信するコネクションを移譲先ノード内の orterun もしくは orted へと通信するように変更する。dmtcp_coordinator に加えた 2 つの実装を以下に示す。

- 各ノードの MPI 管理プロセス (orterun と orted) のアドレス情報を IP アドレスと関連付けて記録する。
- プロセスからの問い合わせに対して通信相手のアドレス情報を発見できない場合、問い合わせ元の IP アドレスを元に、対応する MPI 管理プロセスのアドレス情報を送信する。

この 2 つの実装により、負荷分散されたプロセスは移譲先の MPI 管理プロセスへ接続要求 (connect システムコール) を行う。そしてこの MPI 管理プロセスとの接続を可能にするために、MPI 管理プロセスへと復元される mtcp_restart プロセスにおいて次のような制御を加えた。

- チェックポイント時に記録していなかったプロセスからの接続要求を受け付ける。

接続要求を受け付ける際、復元するソケットファイルディスクリプタとは異なる新しいファイルディスクリプタを用意して受け付ける (accept システムコール)。

また各プロセスにおいて、復元しないコネクション (脱退ノード内の MPI 管理プロセスとのコネクション) は復元

せずソケットを閉じるようにする。

3.3.2 ブロッキング通信への対応

図 4 におけるプロセス間通信の再構築時 (Loop3)、タイミングによって connect() システムコールによる接続要求を accept() システムコールで受け付けられず、接続を確立できない場合がある。通常の DMTCP ではチェックポイント時にプロセスが持つ全てのコネクションを記録しているため、再構築に成功したコネクション数がチェックポイント時に記録したコネクション数より少ない場合はブロッキング通信通信へと切り替え、接続を確立させる。しかし負荷分散プロセスと移譲先ノード内の MPI 管理プロセスとの間のコネクションは、チェックポイント時には存在しないコネクションであるため、タイミングによって接続を確立できない場合がある。

この接続を保証するために、負荷分散プロセスと移譲先ノード内の MPI 管理プロセスとの間のコネクションにおいてもブロッキング通信通信へと切り替えるようにする。そのために負荷分散プロセスと移譲先ノード内の MPI 管理プロセスとの間のコネクション数が必要となる。そこで dmtcp_coordinator に次のような実装を加えた。

- リスタート時に各ノードで起動したプロセス数を記録する。
- 各ノード内の MPI 管理プロセスに対して、自ノード内に起動しているプロセス数を伝達する。

自ノード内に起動しているプロセス数は、図 4 の Loop2 において、プロセスと dmtcp_coordinator との間で交換されるメッセージの中に付加して送信し、伝達する。MPI 管理プロセスでは、送られてきたプロセス数を元に自ノードに負荷分散されたプロセス数を導く。この負荷分散されたプロセス数が MPI 管理プロセスと新たに接続するコネクション数 (N とする) である。図 4 の Loop3 において、accept() システムコールで受け付けたコネクションのうち、負荷分散プロセスとの間のコネクション数が N より小さい場合はブロッキング通信通信へ切り替える。これにより、負荷分散プロセスとの間の接続を保証する。

4. 評価

ノード脱退後のリスタートにおいて、負荷分散機能を使用することによって MPI アプリケーションの実行時間を削減し、処理性能の低下を緩和していることを確認する。

4.1 評価方法/環境

3 ノード構成のクラスタにおいて、各ノードで 4 つの MPI 並列プロセスが実行されているものとする。このとき、1 つのノードが脱退した状況を想定する。脱退したノードで実行されていた 4 プロセスを、クラスタに残った 2 ノードに任意のプロセス数割り当て、実行時間を測定する。現在、Android OS 上で DMTCP を動作させる環境を実現できて

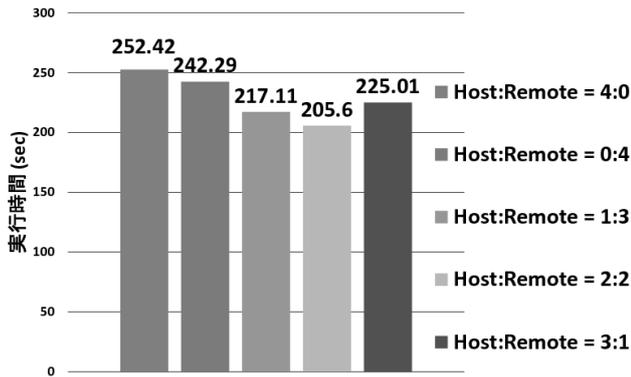


図 6 割り当てプロセス数ごとの実行時間

いないため、Linux PC (動作周波数:3.4GHz, コア数:4, メモリ:32GB) を 3 台利用したクラスタを用いて評価を行う。ノード間通信には 1000Base-T を使用する。評価には MPI 並列版の N クイーンプログラム (クイーン数:19) を使用する。負荷分散機能の効果を明らかにするために、N クイーンプログラムがリスタートしてから解を求めるまでの時間 (実行時間) を比較する。

4.2 評価結果

割り当てプロセス数ごとの実行時間を図 6 に示す。図 6 中の凡例は、dmtcp.coordinator プロセスが起動している PC (Host) と、リモート PC (Remote) それぞれに割り当てたプロセス数である。例えば一番右端のグラフは Host に 3 プロセス (脱退したノード内で起動していた 4 プロセスのうちの 3 プロセス) を割り当て、Remote には 1 プロセス割り当てた場合を示す。また図 6 中の左 2 本のバーがノード単位によるプロセス割り当て、残りの 3 本が負荷分散機能を使用した場合の実行時間を示している。図 6 より、ノード単位によるプロセス割り当てと比較して、プロセス単位で割り当てる負荷分散により実行時間を削減し、処理性能の低下を緩和することができた。性能低下を緩和できた原因は、プロセス単位による割り当てによって起動するプロセス数が減り、1 つのコアを 1 つのプロセスが占有して使用することが可能となったためである。

5. おわりに

我々が開発しているクラスタシステムで MPI アプリケーションのリスタート時においてプロセス単位で並列プロセスを割り当てる負荷分散機能の実装について述べた。プロセス単位での割り当てを可能とするために、DMTCP に主に 2 つの変更を加えた。チェックポイントデータ単体での生成とプロセス間通信の再構築時に、負荷分散プロセスにおける一部の接続 (分散される前に起動していた

ノード内の MPI 管理プロセスとの接続) を委譲先ノード内の MPI 管理プロセスと接続するように変更することで、MPI アプリケーションのリスタートが可能となった。また、変更された接続において接続に失敗した場合、ブロッキング通信へ切り替える制御を加えることで接続を保証した。

同性能の PC を使用した初期評価により、実装した負荷分散機能を使用することで MPI アプリケーションの実行時間を削減し、並列タスクの負荷分散が可能であることを確認した。

今後の予定として、各ノードの処理性能に応じて割り当てるプロセス数を自動で決定する機構の実現が挙げられる。また、ノード間通信を行っていたプロセス同士が負荷分散後、同一ノード内で起動する可能性がある。この時、プロセス間通信の方法の切り替え (ノード間通信を共有メモリ通信へ切り替え) を可能にすることで、通信性能の向上を実現する。

謝辞 本研究は一部 JSPS 科研費 24500055, 15K00068 の助成による。

参考文献

- [1] M. M. Juno, A. R. Bhangwar, and A. A. Laghari, "Grids of Android Mobile Devices", ICICTT, pp.1-3, 2013.
- [2] F. Busching, S. Schildt, and L. Wolf, "DroidCluster: Towards Smartphone Cluster Computing The Streets are Paved with Potential Computer Clusters", In Distributed Computing Systems Workshops (ICDCSW), pp.114-117, 2012.
- [3] G. Hinojos, C. Tade, S. Park, D. Shires, and D. Bruno, "Bluehoc: Bluetooth ad-hoc network android distributed computing", Int. Conf. on Parallel and Distrib. Process. Tech. and Appl. (PDPTA), pp.468-473, 2013.
- [4] A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, V. H. Tuulos, "Misco: A MapReduce framework for mobile systems", PETRA'10 Proceedings of the 3rd International Conference on Pervasive Technologies Re-related to Assistive Environments, Article No.32, 2010.
- [5] 荒井裕介, 大津金光, 横田隆史, 大川猛: "端末の動的な参加・脱退を支援する無線接続型 Android クラスタシステムの実装", 信学技報, Vol.114, No.155, pp.143-148 (CPSY2014-34), 2014.
- [6] Y. Sawada, Y. Arai, K. Ootsu, T. Yokota, T. Ohkawa, "An Android Cluster System Capable of Dynamic Node Reconfiguration", Proc. 7th International Conference on Ubiquitous and Future Networks (ICUFN 2015), pp.689-694, 2015.
- [7] X. Qian, G. Zhu, and X. F. Li, "Comparison and Analysis of the Three Programming Models in Google Android", First Asia-Pacific Programming Languages and Compilers Workshop (APPLC) in conjunction with PLDI 2012, pp.1-9, 2012.
- [8] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop", 23rd IEEE International Parallel and Distributed Processing Symposium, IPDPS2009, pp.1-12, 2009.