ISS16-05

修士論文

動的にノード構成可変なAndroidクラスタにおける負荷分散と効率的通信の研究

2017年3月

澤田 祐樹

宇都宮大学大学院工学研究科
情報システム科学専攻

内容梗概

近年，スマートフォンやタブレットなどの高性能モバイル端末が急速に普及している．これを背景として，モバイル端末は並列分散アプリケーションを実行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．この状況の中で，本研究室では Android OS 搭載のモバイル端末を活用したクラスタシステムを開発している．本システムでは MPI を用いたアプリケーションを実行するためのプラットフォームである．本システムではノードコンピュータとしてモバイル端末を用いているため，ノード構成が動的に変更する可能性があるが，そのような場合においても並列処理を継続するために，並列実行途中の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を可能とする機能を備える．

しかし，本システムにおける従来のリスタート機能では同じノード内の複数の並列タスクを不可分に別のノードへ再配置することしかできなかった．そのためノード構成が動的に変更された場合，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やすかった．加えて，並列プロセス間の通信方法が非効率なままで並列処理を行う問題があった．本研究ではこれらの問題を解決するために，並列タスクの負荷分散を実現する機能と通信の効率化を実現する機能を実装した．これら二つの機能により，ノード構成変更後も効率的な並列処理が継続可能とする．

本論文では 2 つの機能の実装方法を示し，またその機能の効果を，テストアプリケーションを用いて述べる．N-queen プログラムを対象に並列タスクの負荷分散機能を評価した結果，従来のノード単位での再配置に比べて実行時間を 15.6%削減できることを示す．また NPB プログラムを対象に通信の効率化機能を評価した結果，単一ノードでリスタートする状況では実行時間を最大 42.2%削減できることを示す．

A Study of Load Balancing and Efficient Communication in Android Cluster System Allowing Dynamic Node Configuration

Yuki Sawada

Abstract

In recent years, high-performance mobile devices such as smart phones and tablet devices spread rapidly. Based on this background, they have attracted attention as a new platform for parallel and distributed applications. In this situation, we are developing a cluster computer system using mobile devices running Android OS. This cluster computer system is a platform that executes applications using MPI framework. Since this cluster computer system use mobile devices as node computer, node configuration may change dynamically. Even when this case is occur, this cluster system have implemented a mechanism in order to continue parallel processing. In the middle of parallel processing, this mechanism acquire checkpoint data and restart parallel processing from the checkpoint data.

However, the conventional restart function can only relocate atomically multiple parallel tasks in a node to another one node. Therefore, when node configuration dynamically changes, the load of parallel tasks imbalance occurs easily among the nodes in the cluster. Additionally, there is a problem that parallel and distributed applications restart as the communication method between the parallel processes is partially inefficient. To solve these problems, I have implemented a function to realize load balancing of parallel tasks and a function to realize efficiency of communication. By applying these two functions, I make it possible to continue efficient parallel processing even after changing the node configuration.

In this paper, I show how to implement the two functions, and evaluate the effect of their functions using test applications. I evaluated the load balancing function of parallel tasks using N-queen program. I show that the execution time can be reduced by 15.6 % compared with conventional relocation per node basis. I also evaluated the efficiency function of communication using NPB program. In the situation of restarting with a single node, I shot that the execution time can be reduced by 42.2 % maximum.

目 次

第1章　はじめに

近年，Android や iOS に代表されるスマートフォンやタブレット端末等のモバイル端末が急速に普及している．2016 年の段階で世界のモバイル端末の出荷台数は約 19 億 6,000 万台，2022 年まで増加を続け，約 24 億台の出荷が予想されている [1]．これらのモバイル端末の多くが，1 つの CPU パッケージ内にプロセッサコアが複数存在するマルチコアを搭載している．そのため端末単体の処理性能も向上し，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 [2]．また，処理性能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見られる．

高性能なモバイル端末がこのように普及していることを背景として，モバイル端末を並列分散処理のための新たな計算資源として活用することが注目されている [3] [5] [6] [4]．並列分散処理とは 1 つの問題 (処理) を分割し，複数の分散された計算機が処理に必要なデータを交換しながら同時並行的に並列タスクを行うことである．複数台の計算機を活用した並列分散処理によって高い演算性能を得ることが可能である．

この時，複数の計算機を結合して協調動作させ，ひとまとまりのシステムとして扱えるようにしたシステムをクラスタシステムと呼ぶ．またクラスタシステム上で並列分散処理するアプリケーションを並列分散アプリケーションと呼んでいる．クラスタシステムでは並列分散処理を行うために，計算機間で相互接続する必要がある．システムの構成計算機をモバイル端末とした場合でも，モバイル端末が備えている Wi-Fi や Bluetooth 等の無線通信機能を活用することで相互接続は可能である．そのため，モバイル端末を計算資源としたクラスタシステムを構築することは実現可能である．多くのモバイル端末が集まるような機会において，モバイル端末が持つ無線通信機能を利用して複数端末でクラスタシステムを構築することにより，いつでもどこでも高い演算能力を得ることが可能である．1 台のモバイル端末では時間がかかるアプリケーションでも，クラスタシステム内のモバイル端末を活用して並列分散処理することで，実行時間を短くしてアプリケーションを処理できる．

先行研究 [7] では近年のモバイル端末の高性能化と普及率の高さに着目し，モバイル端末を活用して並列分散アプリケーションを実行するクラスタシステムを開発している．クラスタシステムの構成要素であるモバイル端末は，Google がスマートフォン用の OS として提供している Android OS を搭載したモバイル端末を使用している．Android OS には 2 つの大きな特徴がある．1 つ目は普及率である．OS 別のシェアでは，Android OS が世界市場で高いシェアを獲得している (2016 年 3 期時

点:87.8% [8]). 普及率が高いため，クラスタシステムの構築の際に多くの端末を集めやすい環境となっている．2 つ目は開発環境である．また Android OS は仕様が公開されているオープンソースであり，ユーザーが独自機能の追加もしくは編集が可能である．上記の特徴に加えて，我々が Android OS 搭載の端末を活用する理由は 2 つある．1 つ目は Google が提供している開発キットである Android NDK を用いることでネイティブコードを使用できる点である．2 つ目は Android OS のカーネルは Linux ベースである点である．そのため，Linux PC 上で動作するプログラムを Android OS 搭載の端末上で動作できるのではないかと考えた．これら 2 つの理由から本システムでは，Android OS 搭載の端末をシステムの構成要素としている．

また我々以外にも並列分散アプリケーションのための新たなプラットフォームとしてモバイル端末を活用する研究事例もあり，システムの構成にはそれぞれ異なる特徴が見られる [3] [4] [5] [6] . M. M. Juno らは高い演算能力を必要とする研究機関 (大学) において，学生や従業員が持つアイドル状態のモバイル端末を Wi-Fi，もしくは WiMAX により相互接続しグリッドコンピューティングに活用し，大学等の研究機関における科学分野の計算に役立てることを目的としている [3]. グリッドコンピューティングとは「複数の計算資源をネットワークで接続・共有し，ユーザーがそれらを仮想的な巨大コンピュータのように利用できるようにするためのインフラストラクチャ」である [10]. そのため異なるクラスタシステムさえも接続して，巨大なクラスタを構築したシステムもグリッドコンピューティングである．研究 [3] では高い演算性能を得るために演算に必要な資源を共有するプロセッシンググリッドと呼ばれるアプローチを取っている．また Ketan B. Parmar らはボランティアグリッドコンピューティングにおいて，デスクトップコンピュータのみならずモバイル端末も計算資源として活用可能とする独自のフレームワーク (jUniGrid) を提案した [4]. ボランティアコンピューティングとは計算機の持ち主であるユーザーが計算機を利用していない間の余剰計算能力を集めて，大規模な問題を解く方法 [9] であり，安価に利用できることが利点である．さまざまな科学分野 (気候予測や新薬の開発等) で利用可能であり，日本でも千葉県がんセンターと千葉大学が連携し，難治性の小児がんである神経芽腫に対する新しい治療薬を発見することを目的としてボランティアコンピューティングを活用したプロジェクトを開始している．研究 [4] では，特定のシチュエーションにおいてモバイル端末をグリッドコンピューティングシステムに提供することは有効であると主張している．例えば小規模なグリッドコンピューティングにおいてモバイル端末を計算資源として提供した場合，デスクトップコンピュータを計算資源として提供した場合と比較すると小さいが性能向上が示された．また物理的/財政的に拡張が困難なグリッドコンピューティングにおいては，モバイル端末を計算資源として提供することはオンデマンドかつ価格を抑えて，容易に処理性能を向上可能であると示した．

一方，計算資源がより制限された環境において，グリッドコンピューティングよりも小規模であるクラスタシステムを構築する研究もある．タスク/データを処理する演算能力に加えて，戦場のよ

うな計算資源や電力が限られる環境ではバッテリ寿命を節約する必要があるため，Wi-Fi に比べて省電力である Bluetooth を通信手段としたモバイルクラスタシステムを G. Hinojos らは提案している [5]．F. Busching らもモバイル端末をクラスタコンピューティングに活用し，特定のシチュエーションにおいてはサーバの代用として低コストで実現可能であり，今日のボランティアコンピューティングにおける計算資源と見なすことができると主張している [6].

またこれら研究 [3] [4] [5] [6] における並列分散処理基盤も多様である．研究 [3] はグリッドコンピューティングを実現するミドルウェアとして一般的な Globus Toolkit を使用している．Globus Toolkit は活用可能な複数の CPU やメモリなどのリソースを仮想化してユーザーに提供でき，利用者は Globus Toolkit が規程するプロトコルを理解するアプリケーションを開発することでリソースを活用できる [10]．研究 [4] ではグリッドコンピューティングシステムを構築するために独自にプラットフォーム (jUniGrid) を開発した．jUniGrid が提供する API を用いて，計算資源にタスクを割り当てるアプリケーションと，タスクを処理するアプリケーションを開発すればグリッドコンピューティングを行える．本システムと研究 [5] [6] はアプリケーションの並列分散処理に一般的に用いられる MPI[11] を使用している．MPI は並列分処理時，ノード間で処理に必要なデータを交換する方法の規格であり，MPI が提供する関数を用いたアプリケーションを開発する．そして MPI 並列化されたアプリケーションを実行すると，タスク (MPI 並列実行プロセス) 間でデータを交換しながら並列分散処理を行う．

本システムや研究 [3] [4] [5] [6] のように，高い演算性能の実現のためにモバイル端末をクラスタ/グリッドの構成要素に活用する場合には，モバイル端末の移動 (参入や脱退) を考慮したシステムを構築する必要がある．なぜならアプリケーションの処理途中に計算ノードであるモバイル端末が脱退した場合は，処理の継続が不可能となり，最初からアプリケーションを処理し直すというオーバーヘッドが生じる．またアプリケーションの処理途中にモバイル端末がクラスタ/グリッドに新規の計算ノードとして参入した場合，新たな余剰計算能力として活用することでさらなる性能向上を実現する機会が得られる．関連研究において，研究 [5] [6] は，モバイル端末の移動によるノード構成の動的変更には対応していない．研究 [5] では並列処理の負荷分散のために，モバイル端末の脱退と参入を可能とするタスクスケジューリングを課題として挙げている．研究 [6] ではモバイル端末が充電中など脱退の可能性が低いシチュエーションを想定している．通信が不安定な環境下での通信途絶による脱退等，ノード構成の動的変更時において処理を継続する機能は備えていない．MPI を使用している研究 [5] [6] ではどちらも並列処理中にノードが脱退した場合，アプリケーションは中断する．研究 [6] はモバイル端末の脱退や参入には対応せず，計算ノード間の通信を確実にする方針をとった．そこで研究 [6] では無線接続 (Wi-Fi) では接続を保証できないため，有線を使用したモバイルクラスタも提示し，モバイル端末の充電時に有線経由で余剰計算能力として活用することが有効であると主張している．

一方，研究 [3] [4] はモバイル端末の移動によるノード構成の動的変更に対応するためのメカニズムを備えている．確実にノード構成の動的変更に対応するためには，アプリケーションの処理中も計算ノードを把握するメカニズムと処理を継続するためのメカニズムの 2 つが必要である．研究 [3] [4] ではノードの接続状態を確認し，計算資源として活用できるノードと判断した場合にタスクの割り当てを行っている．このように研究 [3] [4] は計算ノードを把握するメカニズムは備えているが研究 [3] は処理を継続するためのメカニズムを備えていない．研究 [3] ではモバイル端末の電池残量で脱退の可能性を予測しているため，通信途絶によってタスクを割り当てたノードが脱退し，タスクが未処理となった場合の対処を用意していない．研究 [4] は割り当てるタスクのステータスを管理する機能を持ち，タスクを割り当てたノードの脱退を検知した場合はそのタスクは未処理であると判断し，別のノードにタスクを割り当て，処理を継続することは可能である．

本システムでもノード構成の動的変更に対応するために，並列分散処理に活用できる計算ノードを把握するメカニズム (ノード管理メカニズム) と処理を継続するためのメカニズム (並列分散処理継続メカニズム) の 2 つを備えている．ノード管理メカニズムは処理対象の並列分散アプリケーションの実行中ではない状態における機構であり，ノード間のメッセージ交換により，並列処理に活用しているノードの生存確認/管理を実現する．並列分散処理継続メカニズムは並列分散アプリケーション実行中の状態における機構であり，ノード構成の動的変更時においても並列分散処理の継続を実現する．研究 [4] では並列分散処理の継続は可能ではあるが脱退したノードが担当していたタスクに関しては処理の最初からやり直す必要があるため，このオーバーヘッドを解消するために本システムではチェックポイント技術を導入する．チェックポイント技術とはプログラムの処理中に発生した障害/中断から復旧するための技術であり，アプリケーションのチェックポインティング処理とリスタート処理の 2 つの処理から成る. チェックポインティング処理ではアプリケーションプロセスの状態を保存し，リスタート処理では保存された状態をもとにプロセスを復元し，アプリケーションを再開できる．アプリケーションの中断前にチェックポインティング処理を完了していれば，処理が中断した場合においても処理の最初からアプリケーションを実行し直す必要はない.

本システムでは，並列分散処理環境におけるチェックポインティング処理とリスタート処理が可能な Distributed MultiThreadedCheckpointing (DMTCP)[12] と呼ばれるチェックポインティングソフトェアを使用する．DMTCP はチェックポインティング処理により，並列分散処理を行う複数のプロセスそれぞれの状態を保存したチェックポイントデータを作成する．これらのチェックポイントデータを用いてリスタート処理を行い，並列分散アプリケーションを再開する．ノード構成の動的変更に対してはチェックポイントデータの移譲により対応する．あるノードが脱退した場合，脱退したノードで生成されたチェックポイントデータをクラスタ内の任意の 1 ノードに送信し，リスタート処理を行うことで，チェックポイントデータを受信した 1 ノードが脱退したノード内で行われた並列分散処理を引き継いでアプリケーションを再開できる．またリスタート処理時，新規参入ノードが

あれば脱退したノードで生成されたチェックポイントデータを新規参入ノードへ送信し，リスタート処理を行うことで，脱退したノード内で行われた並列分散処理を新規参入ノードが引き継いでアプリケーションを再開することも可能である．

ここまでに述べたように，関連研究と比較して，先行研究である本システム [7] はノード管理メカニズムと並列分散処理継続メカニズムを備えているため，クラスタを構成するノードコンピュータとするモバイル端末を把握し，かつ脱退による並列分散アプリケーションの中断が発生した場合でも，アプリケーションを処理の最初から実行し直すことなく並列分散処理を継続可能である．

しかしノード構成の動的変更に伴い並列処理をノード間で移譲する際，ノード間の並列処理の負荷に偏りが生じ，システム全体の性能が大きく低下する場合がある．並列処理の負荷に偏りが生じる原因は，DMTCP における並列処理の移譲はノード単位で行われるからである．例えば脱退したノード内で行われた並列処理を複数のノードに分散して移譲することができない．そのため脱退したノード内で行われた並列処理を引き継いだ 1 ノードだけ負荷が増加する．並列処理を引き継がなかったノード内の並列処理が終了したとしても，負荷が増加したノードの並列処理を待機するため，アプリケーションの実行時間が長くなる．このような並列分散処理の負荷分散の問題以外にも問題がある．DMTCP のリスタート処理において，復元した並列分散処理を行うプロセス間の通信方法が非効率なままで並列処理を行うことである．例えば並列処理 (並列プロセス) の移譲を伴ってリスタートする際，ノード間通信により通信を行っていた並列プロセス同士が同一ノードで復元される場合がある．DMTCP はチェックポイント処理時に記憶したプロセス間通信を再現する仕様であるため，プロセス同士が同一ノードで起動しているにも関わらず，ノード間通信で通信を再現し，通信効率が低いまま復元される．

そこで本研究では先行研究において採用している DMTCP に変更を加え，次の 2 つの機能を実現する．1 つ目は並列処理の移譲時に発生する並列処理の負荷の不均衡を解消する機能である．従来の DMTCP では並列処理の移譲，つまり再配置はノード単位であったため，より粒度の小さいプロセス単位で再配置する機能を実現する．2 つ目はプロセス間通信の効率化する機能を実現する．DMTCP のリスタート処理時，並列プロセス間の通信方法をより効率の良い通信方法に切り替えることでプロセス間通信の効率化を実現する．ノード構成の動的変更に対応し，かつ並列分散処理の負荷分散とプロセス間通信の効率化を実現する機能は先行研究 [7] と関連研究 [3] [4] [5] [6] のどちらにもない新規なものである．この 2 つの機能により，ノード構成の動的変更時においても最大限の性能を維持可能なクラスタシステムを実現できると考える．

ノード構成の動的変更時においても最大限の性能を維持可能なクラスタシステムにより，身の回りで高性能なサーバコンピュータが利用できない状況，あるいはサーバとの間の通信帯域が乏しいシチュエーションにおいて，高性能なサーバコンピュータの代用システムとしての利用が期待できる．具体的には災害時等，計算資源等リソースに制限がある場合に本システムを用いることで，一

時的にサーバの代用として活用できる．またクラスタシステムを構成する端末数が増加すると処理性能が向上する特性からアミューズメントやイベント系のアプリケーションにも応用可能であると考える．例えばチーム対戦型のゲームにおいて，よりユーザー数が多いチームでは処理性能が高く，有利にゲームを進めることができるシステムなどが考えられる．災害時におけるサーバの代用やアミューズメントやイベント系のアプリケーション，これらの用途に共通することはモバイル端末の各所有者が共通の課題/目的を持ち，ボランティアコンピューティングのようにユーザーが協力する体制である．そのため，先に示した 2 例の他にも，複数のユーザーが共通で抱える課題解決/目的達成を図るシチュエーションにおいて本システムの活用が期待できる．

本論文は，次のような構成となっている第 2 章では先行研究であるモバイル端末を用いたクラスタシステム [7] および，現状の課題を述べる．第 3 章ではプロセス単位による並列分散処理の負荷分散の実現方法，第 4 章ではプロセス間通信の効率化の実現方法についてそれぞれ述べる．また一般的な DMTCP の仕様を説明しつつ，2 つの新機能を実現する上での要件と課題について述べる．そして 2 つの新機能，並列分散処理の負荷分散機能と通信の効率化機能の実現方法を述べる．第 5 章は実現した 2 つの機能が効率的な並列分散処理の維持を実現できる機能であるか評価を行う．評価ではノード構成の動的変更後の並列分散処理が効率的か判断するために，実現した機能を使用した場合と使用しない場合のテストアプリケーションの実行時間を比較する．最後に第 6 章で本論文をまとめ，研究により得られた成果を示す．

第2章　Androidクラスタシステム

本研究室では，近年のモバイル端末の高性能化に着目し，モバイル端末を活用して並列分散アプリケーションを実行するクラスタシステムを開発している [7]．また本システムはクラスタシステムを構成するモバイル端末が並列処理中に変更した場合においても，並列処理の継続が可能である．本章では本クラスタシステムの概要と各機能の詳細について述べる．

2.1　システム環境

モバイル端末を活用してクラスタシステムを構築する場合，デスクトップ PC を用いて一般的なクラスタシステムを構築する場合と異なり，特にモバイル端末の移動 (参入や脱退) を考慮したシステムを構築する必要がある．関連研究においても目的や制約によって様々なクラスタシステムシステムデザインが存在する [3] [5] [6] [4]．本システムにおいては静的ノード数変更機構と並列分散処理継続メカニズムの 2 つの機構により，モバイル端末の動的変更に対応する．本節では我々が開発しているクラスタシステム [7] において，構成要素であるモバイル端末，通信環境，アプリケーションの実行環境，静的ノード数変更機構と並列分散処理継続メカニズムそれぞれの詳細を述べる．

2.1.1　構成要素と通信環境

本クラスタシステムではモバイル端末が多く集まるような機会における並列分散アプリケーションの実行を想定しているため，普及しているモバイル端末をクラスタシステムの構成要素とする必要がある．そこで世界市場で高いシェア獲得している Android OS (2016 年 2 月時点:87.6% [8]) を搭載したモバイル端末をクラスタシステムの構成要素とする．そのため我々はこのクラスタシステムを Android クラスタシステムと呼ぶ．また Android OS は仕様が公開されているオープンソースであり，Google が提供している開発キット (2.1.2 で解説) を用いて Android OS 搭載のモバイル端末 (以降 Android 端末) 上で動作するさまざまなアプリケーションを開発することができる．研究 [3] においても Android OS がオープンソースであり，研究者/開発者がモバイル端末上で動作するアプリケーションを開発する機会があるという理由から Android 端末をグリッドコンピューティングに活用している．

次にモバイル端末間の通信環境について述べる．複数の計算機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，高速な通信手段を用いる必要がある．(研究 [5] では特に省電力を優先し Bluetooth を用いてモバイル端末同士は相互接続している．) 研究 [3] では Wi-Fi もしくは WiMAX を用いて相互接続している．研究 [6] は Wi-Fi に加えて有線 (USB ケーブル) を用いて相互接続している．研究 [4] が提案するグリッドコンピューティングシステムでは，接続方法は有線と無線どちらでも接続できるが，有線接続はモバイル端末の「モバイル (移動性)」という目的を捨てるため，グリッドコンピューティングに適用することは賢明ではないと主張している．

本システムは Android 端末が持つ 3 つの無線通信手段 (3G/4G/LTE/回線通信，Wi-Fi，Bluetooth) のうち，高速かつ安価で利用できる Wi-Fi をモバイル端末間の無線通信手段としている．しかし本システムは通信手段を無線接続に限定するわけではない．有線による接続が容易または自然なシチュエーションならば研究 [6] と同様に通信手段を有線としてもよい．

2.1.2 アプリケーションの開発方法

Android 端末上で動作するアプリケーションは Android Software Development Kit (SDK) と Android Native Development Kit (NDK) の 2 つの開発環境を用いて開発することができる．SDK での開発でははアプリケーションを Java 言語で記述し，アプリケーションは Dalvik VM と呼ばれると呼ばれる仮想マシン上 (Android 5.0 以降は ART と呼ばれる) で動作する．一方 NDK は Android アプリケーションを C/C++言語で開発する．生成される実行バイナリはネイティブコードであり，仮想マシンを通さずに直接 CPU が実行するので処理の高速化が可能である．X. Qian らは共通のベンチマークを SDK と NDK を用いて開発し，それぞれの性能を比較している [13]．ベンチマークアプリケーションはユーザーからのタッチなどの入力を取得し，入力が互いに作用する N 個のボールの動きをシミュレートする N-body シミュレーションに似たアプリケーションを活用する．評価では SDK と NDK それぞれで開発したアプリケーションを使用し，FPS（フレーム/秒）値を測定する．そしてアプリケーションが最大 FPS 値 (60FPS) を維持できるときのボールの最大数を比較し，SDK が 50 個，NDK が 500 個のボールまで最大 FPS 値を維持できるという結果を示した．別の表現をすると，SDK で開発したアプリケーションではボールが 300 個の場合，5FPS 以下であるのに対し，NDK で開発したアプリケーションでは同じくボールが 300 個の場合，最大 FPS の 60FPS であった．よって研究 [13] は NDK で開発することにより高速な処理を行えると示している．本システムの目的は Android 端末を活用してアプリケーションを高速に処理する高性能クラスタシステムの実現であるため，Android 端末上で動作する並列分散アプリケーションは NDK を用いて開発する．

2.1.3 並列分散処理基盤

並列分散処理基盤について述べる．複数の計算機を相互接続した並列計算機は，そのメモリ構成により 2 つに分類される．複数の CPU から共通にアクセスできるメモリ (共有メモリ) がある並列計算機を共有メモリ型の並列計算機と呼ぶ．一方，各 CPU からそれぞれのローカルメモリ内に直接参照できない構成になっている並列計算機を分散メモリ型の並列計算機と呼ぶ．分散メモリ型の並列計算機では，各 CPU のメモリの中身を参照するためには，メッセージを交換する (メッセージ・パッシング) 必要がある．メッセージ・パッシングの方法は MPI (Message Passing Interface) と呼ばれる規格が一般的に広く使用されている．共有メモリ型の並列計算機は多くのハードウェア量を必要とすることからも，分散型の並列計算機が主流となっている．現在，共有メモリ型並列計算機の並列性は数百並列しかないのに対し，分散メモリ型計算機の並列性は百万並列にも及ぶ [14].

本クラスタシステムは分散メモリ型並列計算機であり，並列分散アプリケーションを処理するには並列分散処理フレームワークが必要となる．研究 [3] はコンピューティングパワーだけでなく，データベースなどの資源も共有できる Globus Toolkit と呼ばれるグリッドを構築するためのオープンソースのソフトウェアツールキットを活用している．研究 [5] は並列処理のために広く使われている MPI を採用しようとした．(しかし Android OS は Android 特有のカスタマイズが加えられ，一般的に採用されている Linux カーネルと異なるため，Android に MPI を移植できていない．) 研究 [6] は Android に MPI を移植し，Linpack と呼ばれるベンチマークプログラムを並列処理し，台数効果を示した．研究 [4] は Ketan B. Parmar らが提案する jUniGrid フレームワークの API を使用して，タスクを配布/管理するアプリケーションと各計算資源においてタスク実行/結果の返信を実現するアプリケーションを開発する必要がある．

本システムでは研究 [6] 同様，並列分散処理のために最も一般的に用いられている MPI を採用する．現在，多くの MPI 実装が存在する．具体的には LAM / MPI や LA-MPI，MPI-2，MPICH が挙げられる．研究 [6] においては MPI-2 が MPI 実装して使用されている．現在では複数ある MPI 実装おいて，各 MPI 実装との間に互換性がない等の理由から各 MPI 実装の機能を完全に実装した新しい MPI 実装 (Open MPI) が開発された [11]．Open MPI はこれまでの MPI 実装の機能を持ち，オープンソースであるため，本システムでは Open MPI を使用する．以降，並列分散処理に使用する計算資源をノードとする．動作例として，Open MPI を MPI 実装として MPI アプリケーションを実行した場合に各ノードで起動するプロセスを図 2.1 に示す.

図 2.1 では，MPI アプリケーションを起動するノードをホストノード，共に並列処理を行うクラスタ内ノードをリモートノードとし，3 台の計算機それぞれで並列処理を担う MPI 並列実行プロセスが 2 プロセスずつ起動している．ホストノードでは orterun，リモートノードでは orted と呼ばれる MPI 並列実行プロセスを生成/管理するデーモンプロセス (以降 MPI 管理プロセス) が起動する．特に

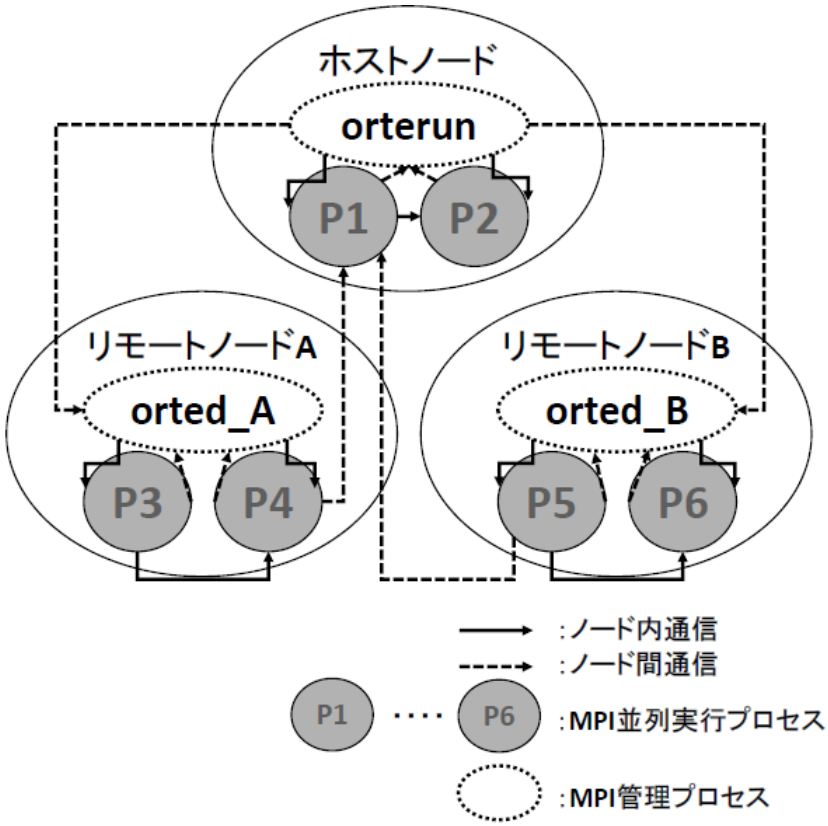

図 2.1　3 ノード 6 並列プロセス時の各ノードの起動プロセス

orterun はホストノード内の MPI 並列実行プロセスの管理だけではなく，リモートノードの orted を管理する役割もある．MPI 並列実行プロセスを生成する orterun もしくは orted は MPI 並列実行プロセスの親プロセスである．各ノードの MPI 並列実行プロセスはプロセス間通信を行い，処理データを送受信しながら並列分散処理を進める．通信方法は一番通信速度が速い通信方法を Open MPI は選択する．ノード内通信の場合のプロセス間の方法は共有メモリを利用したプロセス間通信または TCP を用いたソケット通信となる．ノード間通信は TCP を用いたソケット通信となる．オプションを指定すれば特定の通信方法に限定することも可能である．また，MPI 管理プロセスと MPI 並列実行プロセスとの間には 2 つの通信がある．1 つは MPI 並列実行が接続要求を発行して接続する TCP を用いたソケット通信と，もう 1 つは MPI 管理プロセスが接続要求を発行して接続する UNIX ドメインソケットを用いた通信である．UNIX ドメインソケットを用いた通信は単一ホスト上のプロセス間での通信を実現する．TCP を用いたソケット通信より高速な通信方法である．このように MPI 管理プロセスと MPI 並列実行プロセスとの間には 2 つの通信があるのは Open MPI の仕様である．

また，Open MPI も C 言語で記述されているため，本システムでは高速化のために Open MPI も NDK を用いてビルドする．

2.1.4 ノード管理メカニズム

ノード管理メカニズムはMPIアプリケーションの実行中ではない状態における機構であり，ノードの管理を実現する．本システムのノード管理メカニズムはOpen MPIが用意する‘machine file’と呼ばれるファイルの活用と，ノードの脱退/参入の検知にノード間のソケット通信によるメッセージ交換により，ノードの管理を実現する．

この‘machine file’には並列実行に活用できる各ノードのIPアドレス（またはマシン名）とCPU数を保持している(以降ノードに関するこの情報をホスト情報)．MPIアプリケーションの起動時，このmachine fileを使用して並列分散処理に使用できるノードを特定し，並列処理を行うプロセスを各CPUに割り当てることができる．ノードの脱退が発生した場合はmachine fileから脱退したノードのホストファイルを削除し，ノードが新たに参入した場合はmachine fileに参入ノードのホスト情報を追加する．これによりノードの脱退や参入が発生した場合でも，並列分散処理に活用できるノードを把握することができる．

ノードの脱退/参入の検知について述べる．前述の通り，検知はノード間のソケット通信によるメッセージ交換により実現する．まずノードの脱退の場合は,「いつ，どのノードが脱退するか」を知ることは困難である．そこでクラスタ内の各ノードは，自身が保持するmachine fileに記録されている他のノードに対して，生存確認のためのメッセージを送信する．送信先から応答メッセージを受信できた場合は，そのノードはクラスタ内に生存しているとする．一方で応答メッセージを受信できない場合，ネットワーク通信の一時的な失敗を考慮に入れて生存確認の要求を数回(毎秒)再送する．応答が10回以上ない場合は脱退したと見なし，machine fileから削除する．次にノードが参入する場合，クラスタに参加しようとするノードは自身のホスト情報とともにクラスタ内のすべてのノードに参加要求メッセージを送信する．クラスタ内のノードはクラスタ外のノードから参入要求を受信すると，自身のホスト情報を含む許可メッセージを応答として返信し，参入ノードのホスト情報を自身のマシンファイルに記録する．参入ノードは許可メッセージを受信すると，返信メッセージに付与されたホスト情報をマシンファイルに記録する．

以上により，クラスタはMPIアプリケーションの並列分散処理に活用できるノード(計算資源)を把握できる．

2.1.5 並列分散処理継続メカニズム

並列分散処理継続メカニズムはMPIアプリケーション実行中の状態における機構であり，ノードの動的変更時においても並列分散処理の継続を実現する．MPIアプリケーションの並列分散処理の場合，並列分散処理中にノードの脱退が発生すると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分散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ティング技術を採用する．

チェックポイント技術はチェックポインティング処理とリスタート処理の 2 つの処理から成る．チェックポインティング処理とは実行途中のアプリケーションのバッグアップを作成する．リスタート処理はそのバッグアップを元に，アプリケーションの実行を復元し再開する．このチェックポイント技術を活用し，ノードの脱退が発生する前にチェックポインティング処理を行えば，アプリケーションが中断した場合でもバッグアップをもとに再開し，処理を継続できる．Open MPI には self[15] と呼ばれるチェックポインティング/リスタートのための機能が備わっており，その他にもチェックポイント技術を実現するソフトウェアがいくつか存在する [16] [12]．我々がチェックポイント技術を本システムに導入するに当たり，以下の 2 つが要求される．

1. アプリケーション実行状態を保存でき，保存した状態から処理を再開できること
2. ノードの脱退に対応するために，アプリケーションの実行をノード間で移譲できること

self[15] はユーザレベルでのチェックポインティングである．self によるチェックポインティング機能を利用するためには，アプリケーション内にチェックポインティング処理，リスタート処理をそれぞれおこなうためのコールバック関数を記述する必要がある．さらにどのタイミングでどのような内容 (現在の変数や配列の値) を記録し，またどのような状態から処理を再開するかという情報を MPI アプリケーション毎にユーザが記述する必要があり，開発の負担が非常に大きい．

チェックポインティングソフトェアである Berkeley Lab Checkpoint/Restart (BLCR)[16] は，self がユーザレベルでのチェックポインティングであるのに対し，BLCR はカーネルレベルでのチェックポインティングである．また BLCR はアプリケーション内にチェックポインティングを行うための処理などを記述する必要はない．加えて BLCR ではチェックポイントデータとしてメモリやレジスタの内容等，プロセスに関する全ての情報が記録されるため，ユーザはどのような情報を記録するのかなどを考慮する必要がない．しかし，BLCR はカーネルレベルでのチェックポインティングとなるため，カーネルに依存するため，カーネルが異なる場合に互換性が保たれない．我々は多種多様な Android 端末で構成されるクラスタを想定しているため，カーネルに依存するチェックポインティングは適切ではない．

一方チェックポインティングソフトェアである Distributed MultiThreaded Checkpointing (DMTCP)[12] はユーザレベルでのチェックポインティングであり，x86 や ARM，MIPS の各命令セットに対応している．そして対象アプリケーションを変更することなく，チェックポインティング処理とリスタート処理が可能である．さらにカーネルに非依存であり，並列分散処理をノード間で移譲することが

可能である．以上のような点から本クラスタシステムでは DMTCP をチェックポインティングソフトェアとして採用している．

2.2 DMTCP

DMTCP のソフトウェアアーキテクチャは MTCP レイヤと DMTCP レイヤの二層となっている．MTCP レイヤは個々のプロセス単体のチェックポインティング処理の役割を果たし，DMTCP レイヤは複数ノードに分散している各プロセスにおけるネットワークに関する情報のチェックポインティング処理の役割を果たす．これにより DMTCP は一般的なデスクトップアプリケーションと同様に複数ノード，複数プロセスおよびマルチスレッドからなる並列分散処理のチェックポインティング処理とリスタート処理が可能である．

本節では DMTCP の管理下で MPI アプリケーションを実行した場合を動作例に，チェックポインティング処理とリスタート処理の詳細と動作の流れについて述べる．

2.2.1 チェックポインティング処理

DMTCP の管理下でチェックポインティング処理を有効にするには，実行する MPI アプリケーションの先頭に dmtcp_launch コマンドを付与し，起動する．2.1.2 節で提示した図 2.1 と同様，DMTCP の管理下において 3 ノード 6 並列で MPI アプリケーション実行時の各ノードの起動プロセスを図 2.2 に示す．

図 2.2 より，DMTCP の管理下で MPI アプリケーションを起動するとホストノードには DMTCP の管理デーモンプロセスである dmtcp_coordinator プロセスが起動する．図 2.2 の場合，orterun と orted_A，orted_B，6 つの MPI 並列実行プロセスの計 8 つのプロセスが dmtcp_coordinator プロセスの管理対象となる．また DMTCP 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えて，待機する．チェックポイント取得要求を受け取った場合は自プロセスのチェックポイントを行い，ローカルストレージ内にチェックポイントデータ (以降 ckpt) を作成する．加えてシステムコールを監視する機能 (ラッパー) が管理下のユーザープロセスに追加される．頻繁に呼び出されるシステムコール (read/write) はラップしないが，プロセスの生成やプロセスイメージの変更を記録するために fork() や exec() などのシステムコールを監視する．またプロセス間通信を実現する socket() や connect() などのシステムコールも監視し，プロセス間通信の方法やソケットのタイプまでも記録している．これらのシステムコールをラップし，記録した内容がアプリケーションのリスタート，つまりプロセスの復元時に活用される．プロセスの復元についての詳細は章 3, 4 にて述べる．

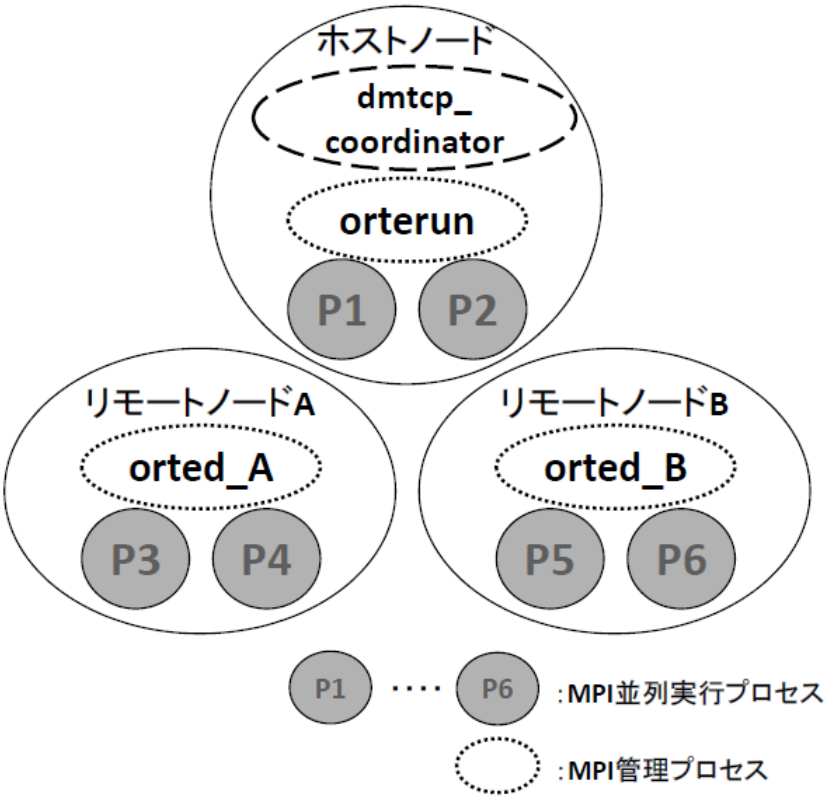

図 2.2　3 ノード 6 並列プロセス時の各ノードの起動プロセス

一般的な DMTCP の活用時において，各 CT がプロセスの ckpt を生成し，プロセスが再開するまでの流れを以下に示す.

STEP0. チェックポイント取得要求の発行:

STEP1. ユーザースレッドの停止:

STEP2. ckpt の生成:

STEP3. ユーザースレッドの再開:

STEP0. では 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 DMTCP 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．STEP1. では CT が dmtcp_coordinator からチェックポイント取得要求を受けとると，プロセス内のユーザープログラムを実行しているユーザースレッドをシグナル (SIGUSR2) により一時的に停止する．STEP2. では CT が自プロセスの ckpt を生成する．STEP3. では CT はユーザースレッドの停止状態を解き，プロセスの実行を再開する．なおチェックポイント処理時にカーネルバッファ内のネットワーク上のデータは，受信者プロセスのメモリにフラッシュされ，ckpt に保存

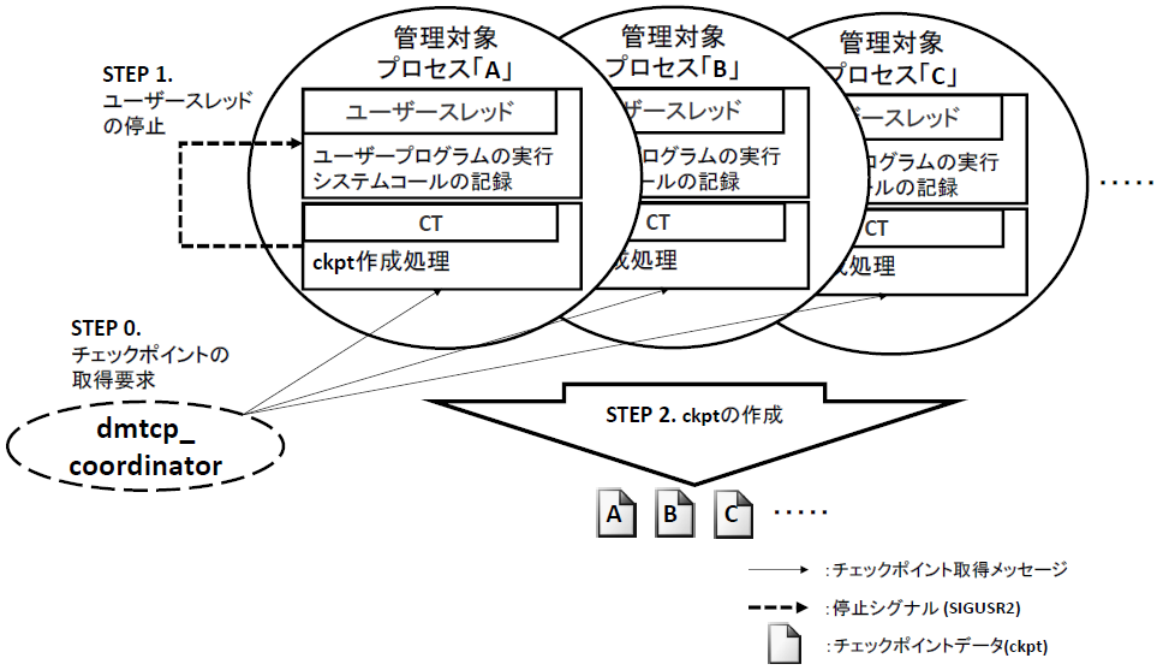

図 2.3　各プロセスの CT による ckpt の生成

する．後に説明するリスタート処理時にはこのネットワークデータは元の送信者に返送され，ユーザースレッドを再開する前に再送信する．dmtcp_coordinator と各 CT が協調してプロセスの ckpt を生成する動作を図 2.3 に示す．

本システムでは ckpt の取得を MPI システム側から制御するために，Open MPI に変更を加えた．

変更を加えた Open MPI ではホストノードの MPI 管理プロセスである orterun が dmtcp_coordinator に対してチェックポイントの取得要求を行う dmtcp_checkpoint() 関数をコールする．チェックポイント関数の呼び出しを MPI 管理プロセスで行う理由は，デーモンによって生成された実際に処理を MPI 並列実行プロセスを処理に専念させるためである．またクラスタ内の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，脱退ノードの並列分散処理を別のノードへ移譲可能とするために，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．なぜならノードの脱退が発生した場合でも，脱退したノード内で起動していたプロセスが全て揃っていれば，クラスタ内に残った別のノードにその ckpt を送信し，並列分散処理の移譲が可能だからである (詳細は 2.2.2 節)．本システムではホストノードが全ての ckpt を持つ．

変更を加えた Open MPI を DMTCP の管理下においた場合，チェックポイント取得からプロセスが再開するまでの流れを以下に示す.

STEP0. チェックポイント取得要求の発行:

orterun が dmtcp_checkpoint() をコールし，チェックポイント取得を要求

STEP1. ユーザースレッドの停止:

STEP2. ckpt の生成:

STEP3. ユーザースレッドの再開:

(STEP4.) (リモートノードのみ) ckpt の送信:

orted は自ノードの ckpt をホストノードに送信し，ホストノードに ckpt を集約

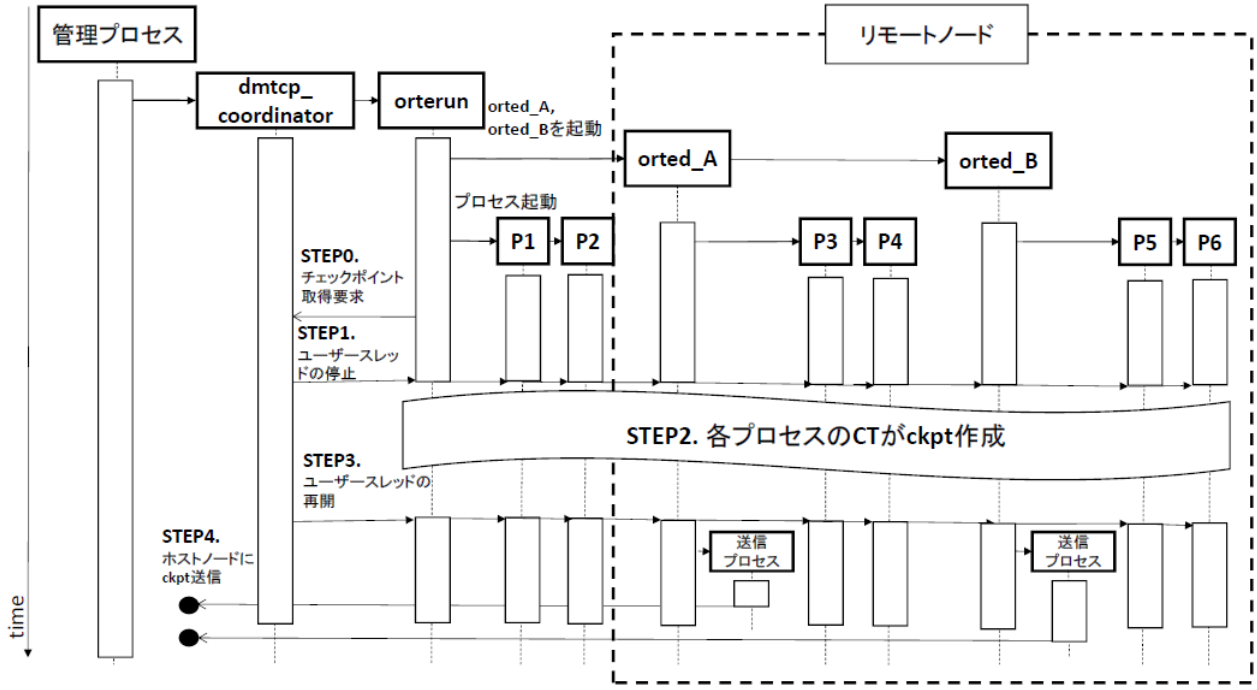

図 2.4　本システムにおける ckpt 取得までの各プロセスのシーケンス

また図 2.2 に示すような 3 ノード構成で構成され，各ノードで 2 プロセスの MPI 並列実行プロセスが起動しているクラスタシステムにおいて，チェックポイント取得からプロセスが再開，加えて ckpt を収集するまでのプロセスの動作シーケンスを図 2.4 に示す．

図 2.4 においては MPI 管理プロセスの orterun と orted_A，orted_B，MPI 並列実行プロセス (P1 から P6) の計 9 プロセスが DMTCP の管理対象となる．リモードノードから送信される ckpt は orted_A，orted_B，P3 P6 の計 6 プロセス分である．ホストノードにはクラスタ内の全プロセスの ckpt があり，仮にリモートノード B が脱退した場合でも orted_B と P5，P6 の 3 つのプロセスをリモートノード A に並列タスクを移譲することが可能である．

以上の ckpt の取得とホストノードへの集約処理を MPI アプリケーションが終了する，または処理の中断が発生するまで続ける．

2.2.2 リスタート処理

DMTCP のアプリケーションのリスタートは，各プロセスの ckpt とチェックポインティング処理時に同時に生成されるリスタート用ヘルパースクリプトを実行することで実現する．

MPI アプリケーションの実行が何らかの理由で中断し，リスタート処理を行うケースは 2 つある．1 つ目はクラスタの構成が MPI アプリケーションの実行開始時と変わらない場合であり，2 つ目はノードの脱退などにより，クラスタの構成が MPI アプリケーションの実行開始時と変わる場合である．クラスタの構成が変わらない場合のリスタート処理の流れは以下となる．

STEP0. ckpt の用意:

各ノードにはリスタートに必要な ckpt がローカルストレージ内に既に用意されている

STEP1. (ホストノード) ヘルパースクリプトの実行:

STEP2. プロセスの復元:

各ノードにおいて，ckpt をもとにプロセスを復元

STEP3. アプリケーションの再開:

一方クラスタの構成が MPI アプリケーションの実行開始時と変化した場合のリスタート処理の流れは以下となる．

STEP0. ckpt の配布:

脱退したノード内の MPI 並列実行プロセスの ckpt を別のリモートノードへ送信

STEP1. ckpt の用意:

各ノードにはリスタートに必要な ckpt がローカルストレージ内に揃う

STEP2. (ホストノード) ヘルパースクリプトの実行:

ホストファイルを活用して，リスタートスクリプトを実行

STEP3. プロセスの復元:

各ノードにおいて，ckpt をもとにプロセスを復元 (脱退ノードの ckpt を受信したノードはその ckpt も復元)

STEP4. アプリケーションの再開:

クラスタ内に残った別のリモートノードが脱退ノードの並列分散処理を継承し，アプリケーションの再開

Step0. では，脱退したノード内で起動していたプロセスを別のリモートノードが継承するために，その ckpt を送信する．Step1. では，脱退したノード内で起動していたプロセスの ckpt もクラスタ内の別ノードに送信され，全 ckpt の復元の準備が整う．Step2. では，DMTCP が提供する機能であるホストファイルを活用してリスタートを行う．このホストファイルでは並列処理を行うノードの指定が可能であり，ノードのホスト名を記述されている．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場合は，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実現できる．Step3. では，各ノードはローカルストレージ内の ckpt をもとに並列タスク (MPI 並列実行プロセス) の復元を行う．特に脱退ノードの並列タスクを継承したリモートノードでは，もともと実行していた並列タスクに加えて継承した並列タスクも復元する．Step4. では，脱退ノードの並列タスクを別のリモートノードが継承し，MPI アプリケーションが再開した．このように DMTCP はノード間で並列タスクの移譲が可能であるため，クラスタのノード構成が変更した場合においても MPI アプリケーションの継続が可能となっている．

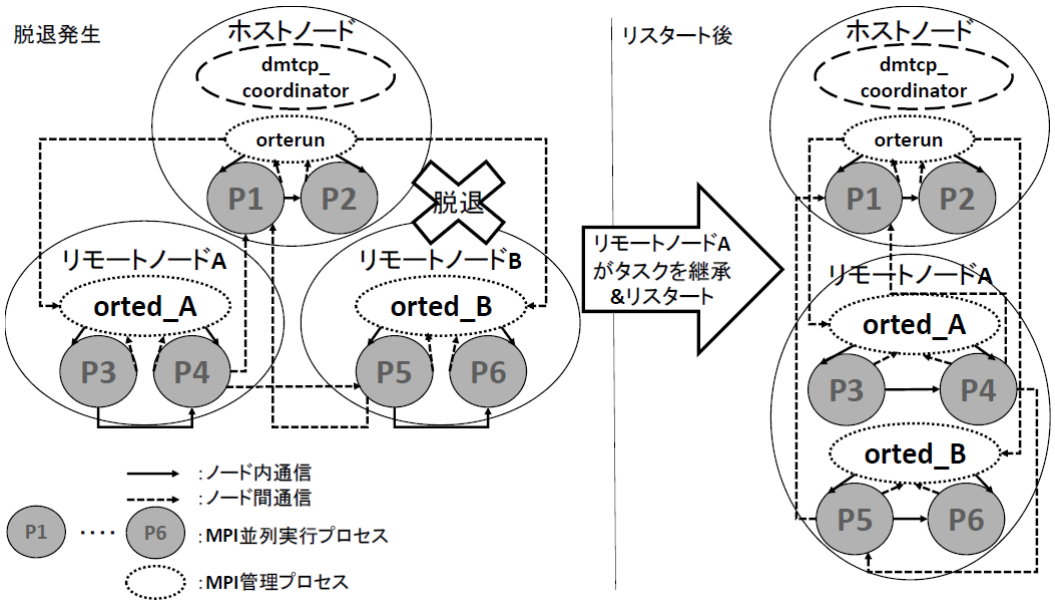

図 2.5　並列タスク継承時における各ノードの起動プロセス

図 2.2 に示すような 3 ノード構成で構成され，各ノードで 2 プロセスの MPI 並列実行プロセスが起動しているクラスタシステムを例に，ノードの脱退時において各ノードで起動しているプロセスを図 2.5 に示す．図 2.5 では 2 つのリモートノード (リモートノード A，リモートノード B) のうちリモートノード B が脱退し，リモートノード A が脱退したリモードノード B の並列タスクを継承する場合である．また図 2.5 に示すように並列タスクの移譲を行う場合のホストファイル内の記述は図 2.6 のようになる．

図 2.6 に示すように，リモードノード B の並列タスクをリモートノード A に移譲する場合はホストファイル内にリモートノード A のホスト名を 2 つ記述する．これにより，図 2.5 に示すようにリモートノード A は脱退が発生する前に自ノードが処理していた並列タスク (orted_A，P3，P4) に加え，それとは別にリモートノード B の並列タスク (orted_B，P5，P6) を復元する．ホストノードは脱退が発生する前に自ノードが処理していた並列タスク (orterun，P1，P2) をそのまま復元する．

```
ホストノードのホスト名
リモートノードAのホスト名
リモートノードAのホスト名
```

図 2.6　並列タスク継承時におけるホストファイル内の記述例

以上が，クラスタの構成が実行開始時と変わらない場合と変わる場合における一般的な DMTCP のリスタート処理の流れである．本システムでは，Open MPI と DMTCP が協調してノードの脱退の検知からリスタート開始のための前処理を行うようにした．その動作の流れを以下に示す.

STEP0. 脱退の検知:

MPI 並列実行プロセスがプロセス間通信の途絶を検知

STEP1. orterun に通知:

シグナルによる通知

STEP2. プロセスの強制終了:

orterun は dmtcp_coodinator 経由で，一度全ての MPI 並列プロセスを強制終了

STEP3. ノードの生存確認:

ノード管理メカニズムによるノードの把握

STEP4. ckpt の配布:

脱退ノード内で作成された ckpt を，並列タスクの移譲先であるノードに送信

STEP5. (ホストノード) ヘルパースクリプトの実行:

ホストファイルを活用して，リスタートスクリプトを実行

STEP6. プロセスの復元:

各ノードにおいて，ckpt をもとにプロセスを復元 (脱退ノードの ckpt を受信したノードはその ckpt も復元)

STEP7. アプリケーションの再開:

クラスタ内に残った別のリモートノードが脱退ノードの並列分散処理を継承し，アプリケーションの再開

STEP0. ではプロセス間通信の途絶の検知は Open MPI がもともと持つ機能であるため，これを活用する．STEP1. ではホストノード内の 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．STEP2. ではリスタートに備えて，一度各ノードの MPI 並列プロセスを強制終了する．dmtcp_coodinator は MPI のプロセス (MPI 管理プロセス含む) を管理対象とし，管理対象のプロセスを全て強制終了する機能を持つため，この機能を活用する．STE3. ではノード管理メカニズム，つまりはメッセージ交

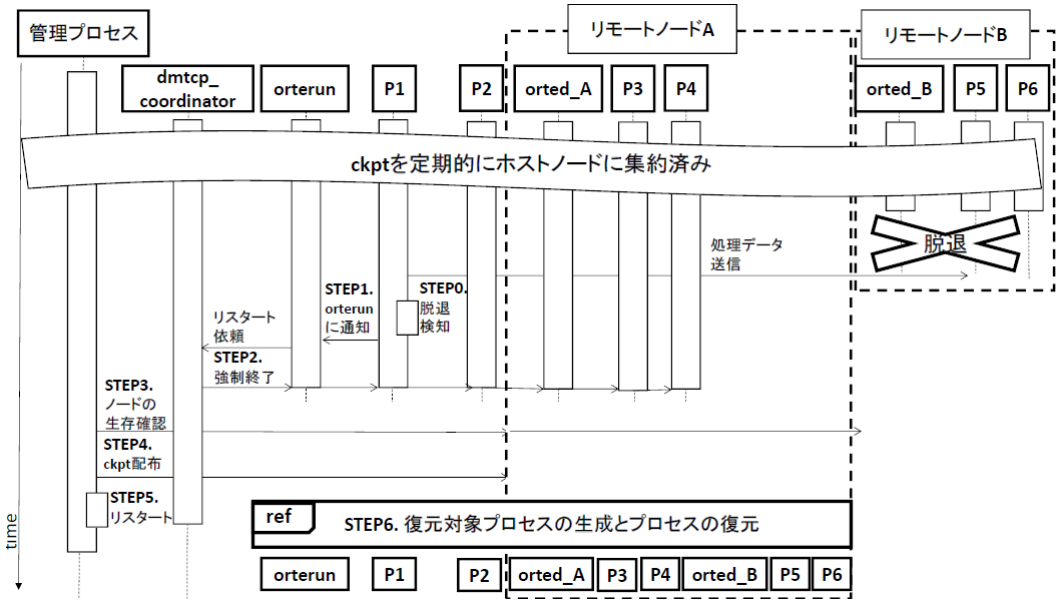

図 2.7　並列タスクの移譲を伴った MPI アプリケーションにおけるプロセス復元までのシーケンス

換によりクラスタ内に残っているノードを確認する．STEP4. ではホストノードに集約した ckpt のうち，脱退ノード内で作成された全ての ckpt を並列タスクの移譲先であるノードに送信する．このときホストノードは並列タスクの移譲先に指定できない．理由は MPI の MPI 管理プロセスである orterun と orted の間の通信に使用する通信用ポートの番号が同じであるため，orterun と orted は同じノード内に起動できない．これは Open MPI の仕様である．同一ノードに複数の orted が起動することは可能であるため，並列タスクの移譲先ノードの選択肢はリモートノードとなる．STEP5. では動作例として示した図 2.6 のようにホストファイルを活用して，ヘルパースクリプトを実行する．そして STEP6，STEP7 を通して並列タスクの移譲を伴った MPI アプリケーションのリスタートが実現となる．上記の動作 STEP をもとに，並列タスクの移譲を伴った MPI アプリケーションのリスタートにおけるプロセス復元 (STEP5.) までのシーケンス図を図 2.7 に示す．

図 2.7 は，図 2.5 のように 3 ノード構成のクラスタにおいて，2 つのリモートノード (リモートノード A，リモートノード B) のうちリモートノード B が脱退し，リモートノード A が脱退したリモードノード B の並列タスクを継承する場合である．(図 2.7 における STEP7. (プロセスの復元) のシーケンスの詳細は 3 で述べる．) 図 2.7 ではリモートノード B のタスクをリモートノード A が継承し，図 2.5 に示すように 6 つのプロセス (orted_A，P3，P4，orted_B，P5，P6) がリモートノード A 上に復元される．ホストノード内のプロセス復元も同時に行われ，全プロセスの復元が完了した後，アプリケーションは再開する．再開後は図 2.4 に示すように ckpt 取得の取得とホストノードへの集約を繰り返す．

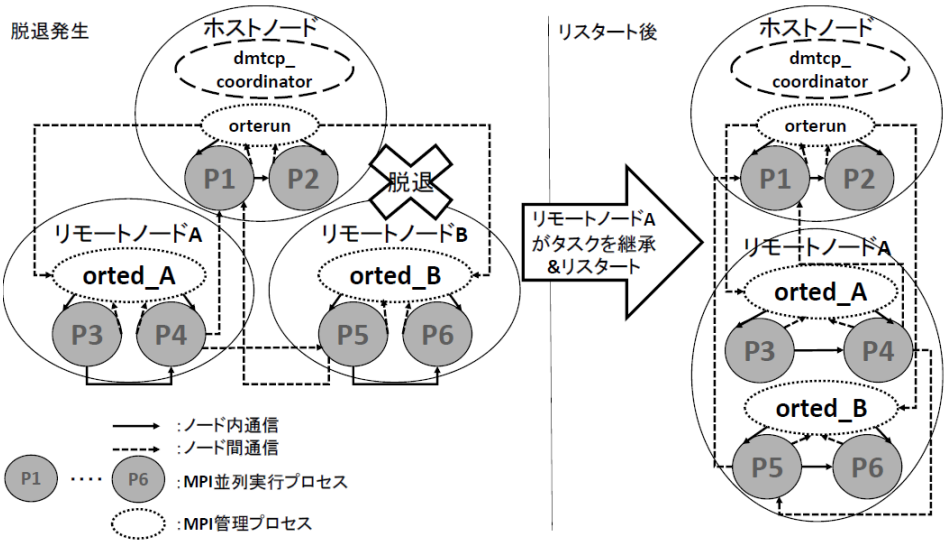

図 2.8　並列タスク継承時における各ノードの起動プロセス (図 2.5:再掲載)

2.3　現状のシステムの課題点

本システムは並列分散処理継続メカニズムを備えているため，ノード間の通信途絶とそれによるアプリケーションの中断が発生した場合でもチェックポイントを取得した地点からアプリケーションを再開できる．しかし MPI アプリケーションの実行を継続可能であるが，ノード数が動的に変更した際にノード間の並列タスクの負荷に偏りが生じ，システム全体の性能が低下する場合がある．加えて通信方法が一部非効率なプロセス間通信が発生する．

はじめにノード間の並列タスク負荷に偏りについて述べる．図 2.7 に示すように，脱退ノードの並列タスクを移譲するには STEP4. において脱退ノード内で作成された全ての ckpt を移譲先ノードに送信する必要がある．このように DMTCP はノード単位で並列プロセスの ckpt を管理し，同時に並列タスクの移譲もノード単位で行う．このノード単位での移譲は DMTCP の仕様である．ノード単位で並列タスクを移譲した場合の例として，図 2.5 を再掲載する．

図 2.8 では 3 ノードで構成されるノードのうち，リモートノード B の並列タスクをリモートノード A へ移譲した場合における各ノード上で起動しているプロセスを示している．DMTCP はノード単位で並列タスクを移譲するため，脱退したリモートノード B の MPI 並列実行プロセス (P5, P6) 全てをリモートノード A のみに移譲することとなる．よって図 2.8 の例ではリモートノード A で起動するプロセス数だけ増加し，並列タスクの負荷がホストノードより大きくなる．

次に非効率なプロセス間通信について述べる．非効率なプロセス間通信とはより速い通信方法があるにも関わらず，遅い通信方法でプロセス間通信が行われることを指す．この非効率なプロセス間通信が発生する例を，既出した図 2.8 を用いて解説する．図 2.8 では脱退したリモートノード B

の並列タスクをリモートノード A に移譲し，リスタートしている．この際復元されるプロセスにおけるプロセス間通信の方法は ckpt 作成時と同じである．2.2.1 節で述べたように，DMTCP はチェックポイント処理にプロセス間通信に関する情報も保存する．そしてノードの移譲の有無に関係なく，チェックポイント処理時と同様のプロセス間通信を再構築する．図 2.8 において，脱退発生前はノード間通信を行っていた MPI 並列実行プロセス P4，P5 は並列タスクの移譲により同一ノード (リモートノード A) 内に復元される．復元時，P4 と P5 は同一ノードであり，より高速なノード内通信の方法があるにも関わらず，通信方法はノード間通信として復元される．

以上のように，本システムにおいて並列タスクの移譲を行う場合，並列タスクの負荷に不均衡が生じる．加えてプロセス間通信の方法が非効率な通信が発生する等の問題がある．そこで本研究ではノード単位ではなくプロセス単位で並列タスクを各ノードに再配置する負荷分散方法を提案する [17]．図 2.8 において，プロセス単位での負荷分散を実現することで脱退したリモートノード B の並列タスク (P5，P6) をホストノードとリモートノード A それぞれに 1 プロセスずつ再配置し，処理性能の低下を緩和できると期待する．また非効率なプロセス間通信が発生するという問題を解決するために，リスタート処理時により高速な通信方法へ切り替える方法を提案する．図 2.8 において，リスタート処理時により高速な通信方法へ切り替えることが可能となれば同一ノード内で起動している P4，P5 間の通信方法を高速なノード内通信へ切り替え，同様に処理性能の低下を緩和できると期待する．

第3章　並列タスクの負荷分散

本章はノードの動的変更時に発生するノード間の並列タスクにおける負荷の不均衡を解消するために，DMTCP に追加した負荷分散機能について述べる．

3.1　負荷分散の概要と要求

私が提案する負荷分散方法は，従来の DMTCP がノード単位で並列タスクの移譲 (以降 再配置) を行うのに対し，プロセス単位で複数ノードに並列タスクを再配置可能とする．

図 3.1 にノード単位で並列タスクの再配置を行った場合とプロセス単位で再配置を行った場合のプロセスの移動を示す．

図 3.1 において，プロセス単位での再配置を可能とすることでより柔軟にノード間における並列タスクの負荷バランスを調整でき，効率的な並列分散処理の維持が可能であると考える．しかしプロセス単位での再配置時，脱退したノード内の orted の復元についてとプロセス間通信の再構築について考える必要がある．

まず脱退したノード内の orted の復元について述べる．プロセス単位で MPI 実行プロセスを複数ノードに再配置時，MPI 実行プロセスの共通の管理デーモンである orted をどう処理するか検討する．図 3.1 を例とし，仮に複数ノードに再配置する MPI 実行プロセス (P1，P2) にそれぞれ付随して orted_A を再配置しようとした場合，本来 1 つしかない orted_A が複数復元する状態となり，MPI アプリケーションのリスタートが正常に動作しないと考える．また orted_A が複数復元する状態を可能とする場合は，DMTCP におけるプロセスの復元処理と Open MPI 側の両方に大規模な変更が必要となり，複雑になる．そこで本研究では orted は復元せずに，再配置される MPI 実行プロセス (図 3.1 の例だと P1，P2) は新たに再配置されるノード先の orterun もしくは orted とコネクションを確立することで，シンプルにプロセス単位での再配置が実現できると考える．そのため図 3.1 中では，プロセス単位での再配置の場合は脱退したノードの orted_A はどのノードにも再配置していない．加えてホストノードに対してもプロセスを再配置可能となる．

次にプロセス間通信の復元について述べる．プロセス単位での再配置時，チェックポイント処理時に記録したプロセス間通信を再構築できない場合が発生する．以下にその発生パターンを示す．

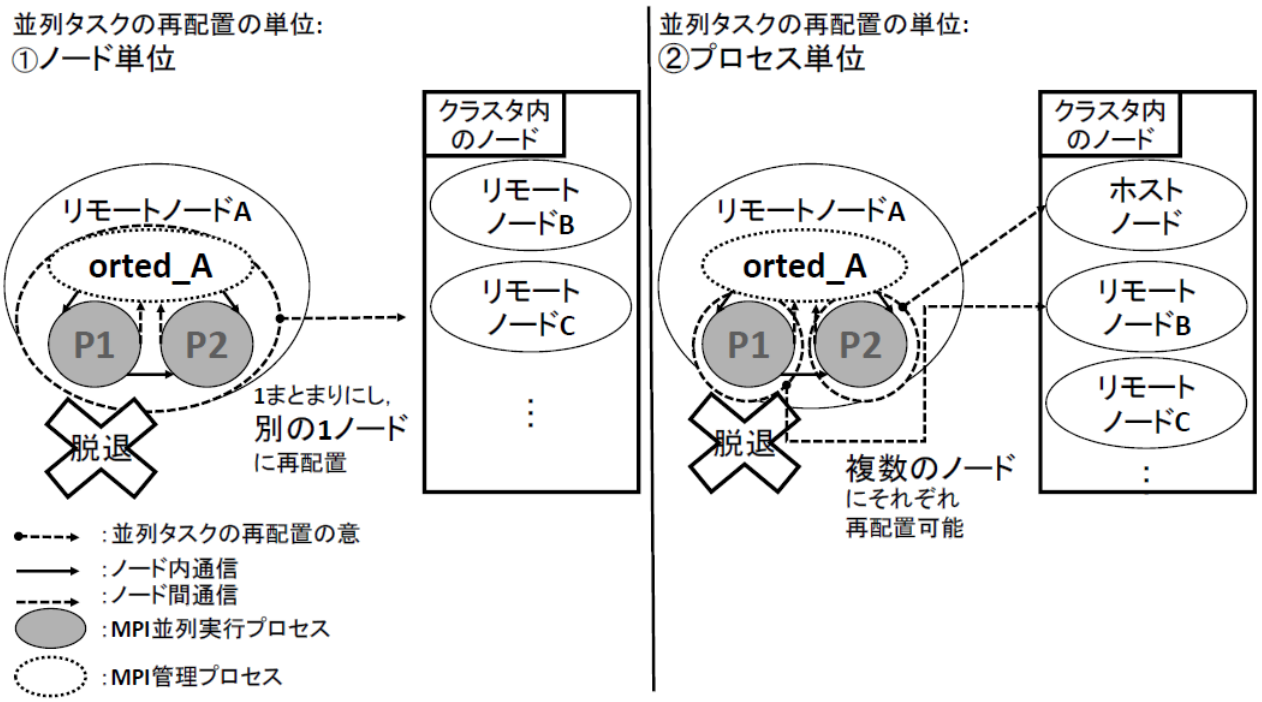

図 3.1 各並列タスクの再配置方法によるプロセスの移動

1. チェックポイント処理時に記録した通信方法では再構築できない場合

2. チェックポイント処理時に記録した通信が行われない場合

1 つ目は，ノードをまたいだプロセスの移動によってチェックポイント処理時に記録した通信方法では通信できない場合である．Open MPI はさまざまな通信方法を提供しており，MPI アプリケーションの起動時において，通常はプロセス間の通信が最速となる通信方法を選択し，通信路を構築する．例えばあるプロセス間の通信が同一ノード内の場合は高速な共有メモリ通信，ノード間通信の場合はネットワークを介した通信 (TCP を用いた通信) となる．図 3.1 中においても MPI 並列実行プロセス P1，P2 は同一ノードであるため，共有メモリ通信となる．チェックポイント処理時には共有メモリ通信として通信方法を記録するが，プロセス単位での再配置によって別々のノードに P1，P2 が再配置された場合，ノード間通信へと通信形態が変更する．しかし DMTCP はチェックポイント処理時の通信方法でプロセス間通信の再構築を試みるため，再構築に失敗し MPI アプリケーションはリスタートすることができない．そのため図 3.1 の例ではプロセス間通信の再構築には失敗する．そこで本研究では MPI 並列実行プロセス間の通信をネットワークを介した通信に統一し，共有メモリ通信を使用しないこととした．全てのプロセス間通信がネットワークを介した通信であるため，プロセスがノードをまたいで移動した場合でもプロセス間通信を再構築でき，MPI アプリケー

ションも再開できる．同一ノード内におけるプロセス間通信もネットワークを介した通信となるため，共有メモリ通信を採用した場合と比べて通信性能は劣るが，プロセス単位での再配置をシンプルに実現できる．また第 4 において，プロセスの再配置時にプロセス間通信をより通信性能が高い方法に変更する機能について述べる．2 つ目は，記録した通信が行われない場合である．本研究ではプロセス単位での再配置をシンプルに実現するために脱退したノード内の orted は復元しない方針としたが，それにより一部通信の断絶が起こる．図 3.1 の例では，プロセス単位での再配置により orted_A と P1 間の通信と，orted_A と P2 間の通信が断絶する．しかし DMTCP はチェックポイント処理時にこれらの通信は記録されているため，プロセス間通信の再構築を試み失敗する．DMTCP がノード単位での並列タスクの再配置が可能であったのは，MPI のプロセスを 1 まとめにしていたため，プロセス間通信の再構築を含めたプロセスの復元に成功していた．そこでプロセス単位での再配置を実現するには，プロセス間通信の再構築時に断絶した通信に関しては再構築を行わないもしくは通信先を付け替える等の処理を含め，プロセス単体での復元も可能とする必要となる．

以上を踏まえて，プロセス単位での再配置を実現するための前提条件と要求を再度図 3.2 に示す．

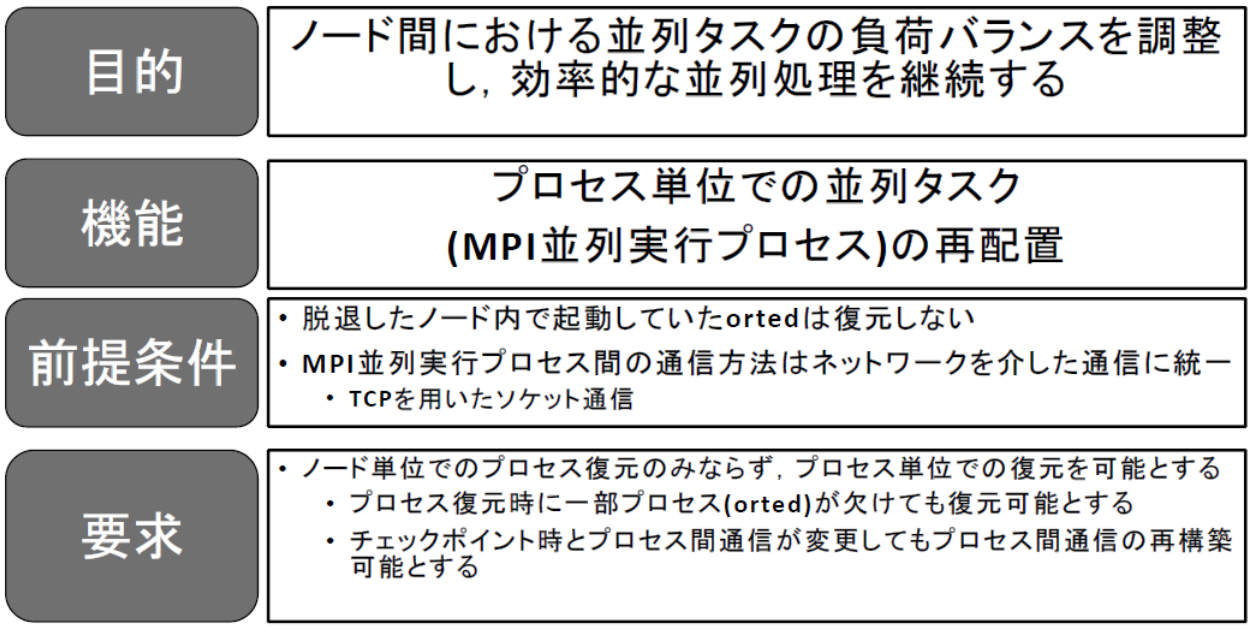

図 3.2　並列タスクの負荷分散実現における要求と要件

3.2　プロセス単位での並列タスクの再配置

3.1 節において，プロセス単位での並列タスクの再配置おける概要と実現するための要求を述べた．本節では要求を実現するために，一般的な DMTCP におけるプロセスの復元処理を解説し，実装上の課題/要件をはじめに提示する．そして要件を満たすために行った実装方法を述べる．私が変更を加えた部分は特にプロセスの復元処理中の ‘復元対象プロセスの生成’ と ‘プロセス間通信の再構築’ の 2 つの処理である．そのため，復元対象プロセスの生成，プロセス間通信の再構築の順で述べる．

3.2.1　一般的な復元対象プロセスの生成と実装上の要件

2.2.2 節において，DMTCP のリスタート処理はチェックポイント処理時に作成したリスタート用ヘルパースクリプトを使用すると述べた．図 3.3 に示すヘルパースクリプト例の場合，ヘルパースクリプト実行後の各ノード内で起動しているプロセスを図 3.4 に示す．

図 3.3 に示すように DMTCP 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．図 3.3 中の一行目はホストノードで復元するプロセスの ckpt(orterun, P1，P2) を指定し，ヘルパースクリプト実行後は図 3.4 に示すようにホストノードでは orterun，P1，P2 の 3 プロセスが復元する．2 行目以降はリモートノードにおいて復元するプロセスの ckpt を指定している．リモートノードの場合は ssh コマンドでリモートログインし，dmtcp_restart コマンドを実行する．図 3.3 において，リモートノード A では 3 つの ckpt(orted_A，P3，P4)，リモートノード B では 3 つの ckpt(orted_B，P5，P6) が復元対象であり，指定通りのプロセスが図 3.4 に示すように復元されている．

dmtcp_restart.sh

```
dmtcp_restart  ckpt_orterun_host  ckpt_P1_host  ckpt_P2_host
ssh remote node A “dmtcp_restart ckpt_orted_A  ckpt_P3_A  ckpt_P4_A”
ssh remote node B “dmtcp_restart ckpt_orted_B  ckpt_P5_B  ckpt_P6_B”
```

図 3.3　3 ノード各 2 プロセス実行アプリケーションにおけるヘルパースクリプト

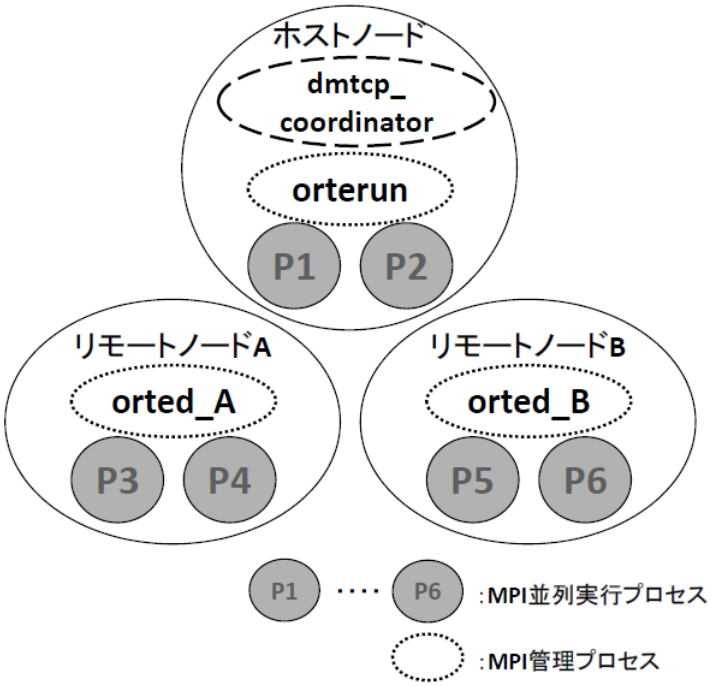

図 3.4　図 3.3 のスクリプト実行後の各ノードの起動プロセス

次に図 3.3 に示すヘルパースクリプト実行から復元対象プロセスの生成までのプロセスの生成/状態遷移シーケンスを示す．図 3.5 がホストノードにおけるプロセスの生成/状態遷移シーケンス，図 3.6 がリモートノード A におけるプロセスの生成/状態遷移シーケンス，図 3.7 がリモートノード B におけるプロセスの生成/状態遷移シーケンスである．プロセスの生成/状態遷移シーケンスにおいて，ホストノードとリモートノードの違いはホストノードでは dmtcp_coordinator を起動し，リモートノードでは起動しない点である．リモートノード A，B におけるプロセスの生成/状態遷移シーケンスを示す図 3.6 と図 3.7 との違いは復元後のプロセスが異なるだけでシーケンスは全く同じである．

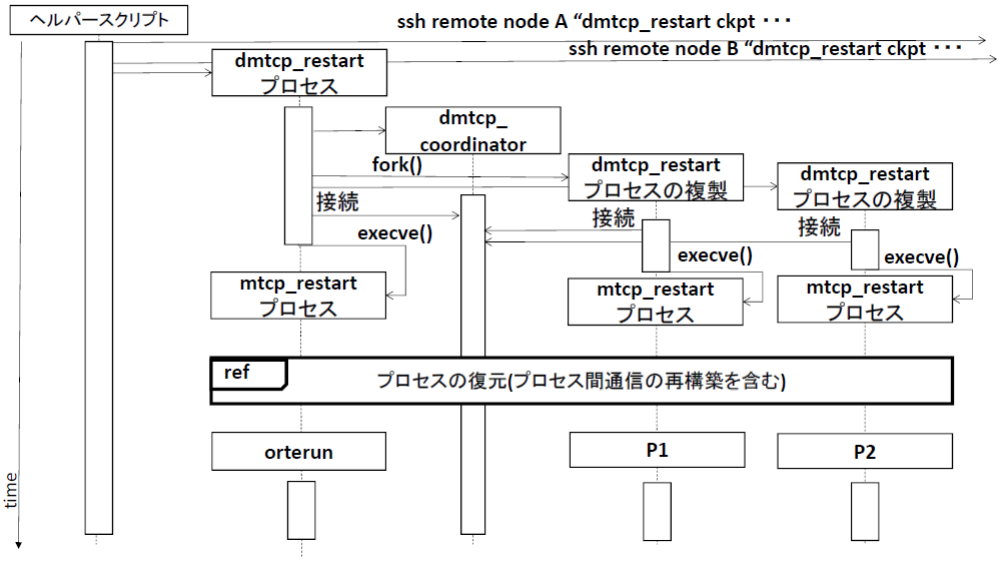

図 3.5　図 3.3 のスクリプト実行時のプロセス生成シーケンス:host node

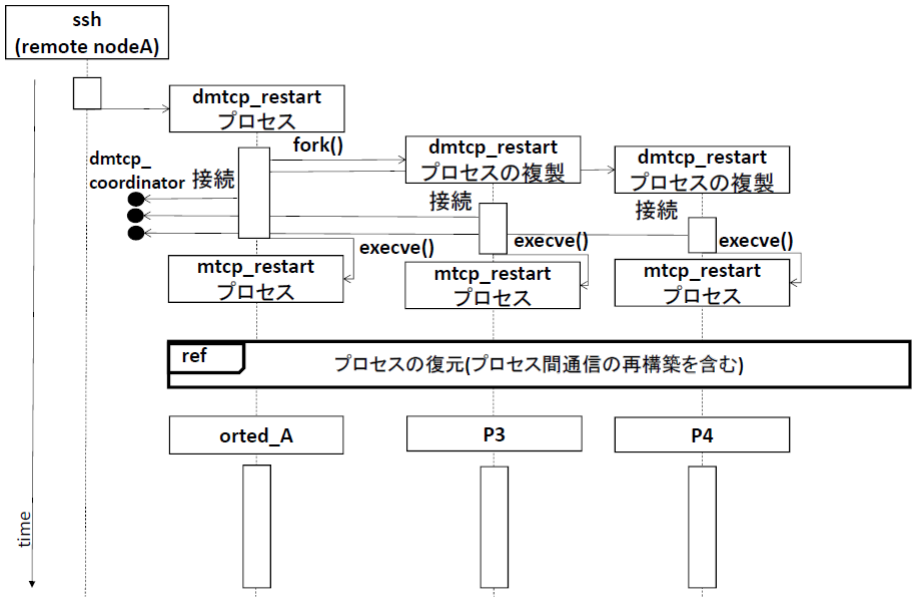

図 3.6　図 3.3 のスクリプト実行時のプロセス生成シーケンス:リモートノード A

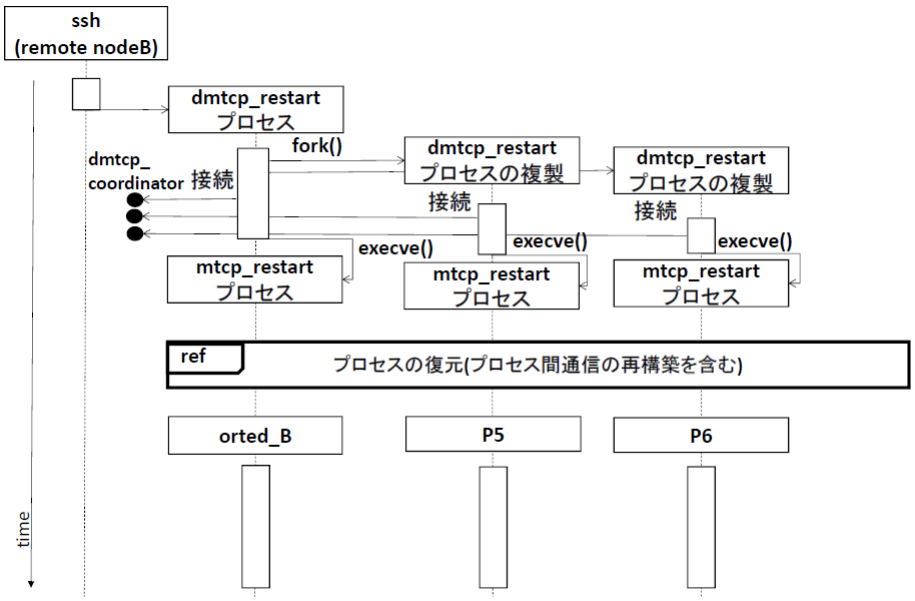

図 3.7　図 3.3 のスクリプト実行時のプロセス生成シーケンス:remote nodeB

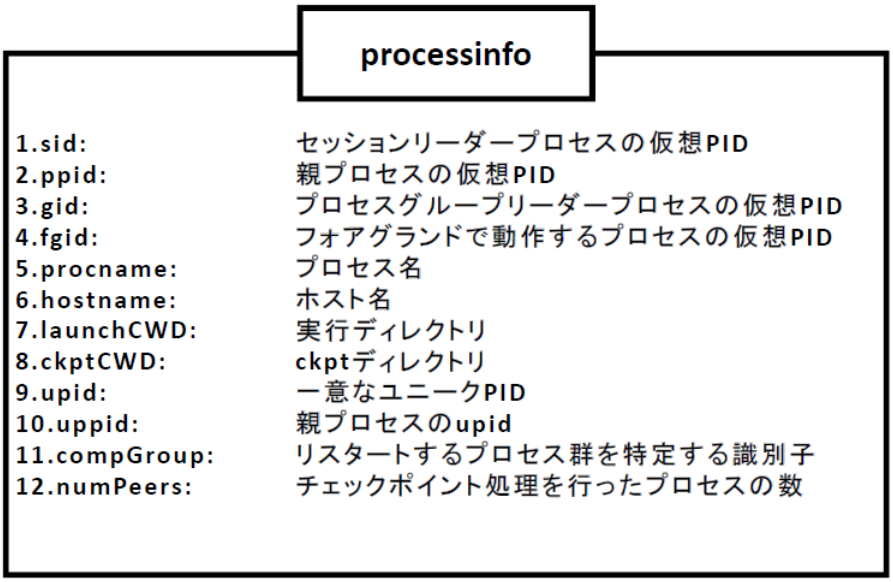

図 3.8　ckpt における主要なプロセスのヘッダ情報

図 3.5，図 3.6，図 3.7 において各ノードでは 1 つの dmtcp_restart プロセスが起動する．1 番最初に起動したこの dmtcp_restart プロセスが復元処理によって MPI の管理プロセス (orterun もしくは orted) となる．復元処理によって親プロセスである MPI の管理プロセスに変化する dmtcp_restart プロセスが最初に起動することは DMTCP の仕様である．そして fork() システムコールにより必要な数分の dmtcp_restart プロセスを複製する．それぞれの dmtcp_restart プロセスの複製は dmtcp_coodinator に対してプロセス名や一意なプロセス識別子をメッセージとして送信する．このメッセージの受け取りにより dmtcp_coordinator はこれから復元/管理するプロセスを把握することができる．その後，dmtcp_restart プロセスの複製は execve() システムコールにより復元処理を行う mtcp_restart プロセスへと変化する．そして復元処理が完了すると MPI プロセスとしてリスタートする．図 3.5 に示すようにホストノード内の mtcp_restart プロセスはそれぞれ orterun，P1，P2 へと復元される．同様に図 3.6，図 3.7 に示すようにリモートノード内の mtcp_restart プロセスはそれぞれ orted_A，P3，P4，orted_B，P5，P6 へと復元される．

前述で，必要な数分の dmtcp_restart プロセスを複製すると述べたがこの複製処理にはプロセス間の親子関係の情報が活用される．fork() システムコールを実行する dmtcp_restart プロセスはあらかじめ引数に取った各 ckpt のヘッダ情報を持つ．ヘッダ情報にはその ckpt が復元するプロセスの親子関係を含む一般的な情報が含まれている．その中で主要な情報を図 3.8 に示す．

DMTCP は管理下のプロセスに仮想 PID を割り当て，仮想 PID を参照して様々な処理を行うため，図 3.8 に示すように仮想 PID を用いた情報を記録する．図 3.8 中の upid はプロセス固有のユニークな ID であり，DMTCP が割り当てる．図 3.8 の情報に加えて子プロセスを持つ親プロセスは子プロ

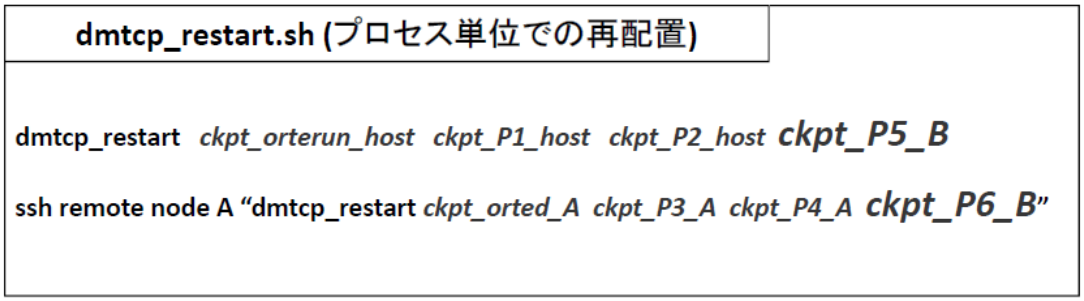

図 3.9　プロセス単位での再配置を行う場合のヘルパースクリプト例

セスの upid を childTable と呼ばれる変数により記録しているため，dmtcp_restart プロセスはこれらの情報を活用し，復元する MPI 管理プロセスと親子関係にある ckpt を特定する．そしてその ckpt を復元対象とし，fork() システムコールを実行する．以上が一般的な DMTCP における復元対象プロセスの生成シーケンスである．

プロセス単位での再配置を可能にするために，本研究では再配置するプロセスの ckpt を再配置先ノードで実行される dmtcp_restart コマンドの引数として与える方法を検討する．図 3.4 に示すように 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に，プロセス単位での再配置を行う場合のヘルパースクリプトを図 3.9 に示す．

図 3.9 は 2 プロセス (P5，P6) 起動していたリモートノード B が脱退し，ホストノードとリモートノード A にそれぞれ 1 プロセスずつ再配置する場合のヘルパースクリプトである．図 3.9 のヘルパースクリプトの実行が成功すればホストノードには 4 つのプロセス (orterun，P1，P2，P5)，リモートノード A でも 4 つのプロセス (orted_A，P3，P4，P6) が復元し，MPI アプリケーションがリスタートする．しかし一般的な DMTCP は，プロセス間における親子関係の有無により復元対象の ckpt を決定する．図 3.9 において，各ノードで実行する dmtcp_restart コマンドに引数として与えた ckpt_P5_B，ckpt_P6_B は再配置先の MPI 管理プロセスとは親子関係にないため，復元対象の ckpt とはならない．プロセス単位での復元を可能にするための要件を以下に示す．

要件 1. 親子関係にないプロセスの ckpt でも復元対象とする

要件 2. 復元に成功する (特にプロセス間通信の再構築)

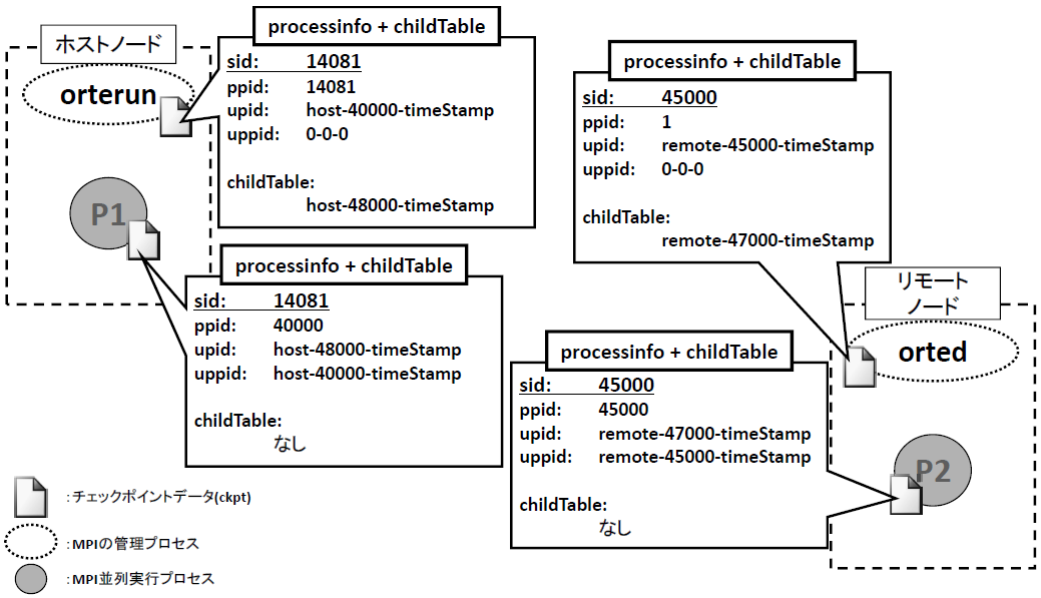

図 3.10　ckpt のプロセスヘッダ情報の具体例

3.2.2　プロセス単位の復元対象プロセスの生成

本研究では DMTCP に変更を加え，親子関係にないプロセスの ckpt でも fork() し，dmtcp_restart プロセスを複製できるようにする．そのために復元対象 ckpt の選定条件を変更する．DMTCP における復元対象 ckpt の選定条件には図 3.8 で示した ckpt のヘッダ情報と子プロセスの upid を記録した childTable 変数が活用される．図 3.10 に，2 ノードで構成され各ノード 1 プロセス MPI 並列実行しているクラスタにおいてチェックポイント処理を行った場合のプロセス情報と childTable 変数の値を例示する．

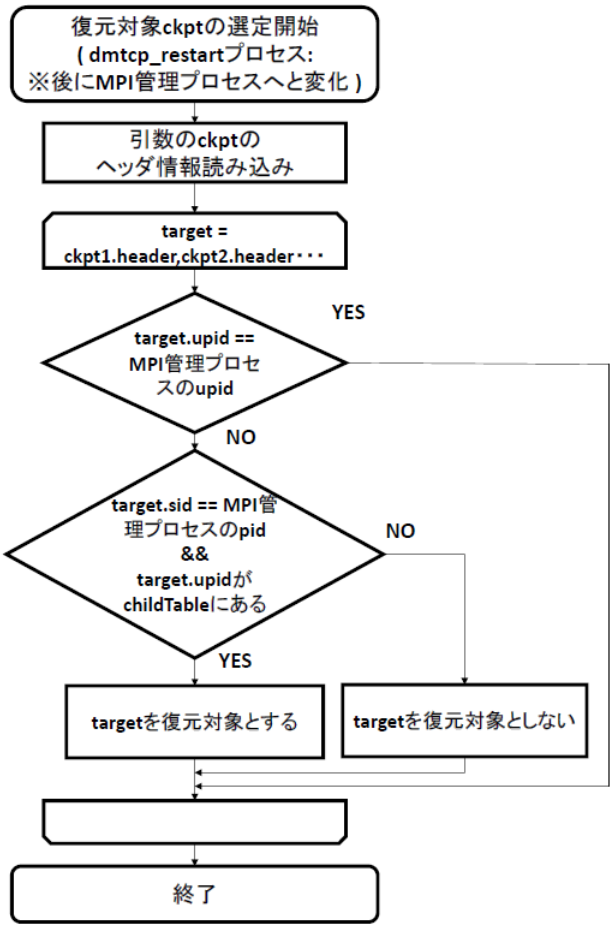

図 3.11　DMTCP における復元対象 ckpt の選定フローチャート

図 3.10 に示すように MPI 管理プロセスとその子プロセスは共通の sid を持ち，この sid はノードごとに異なる．親プロセスである MPI 管理プロセスは子プロセスの upid を childTable 変数に記録している．DMTCP は主に sid と upid，childTable 変数を活用して復元する ckpt を選定する．一般的な DMTCP における復元対象 ckpt の選定フローチャートを図 3.11 に示す．

図 3.11 のフローチャートより復元対象 ckpt の選定条件を述べる．最初の条件分岐で target.upid が MPI 管理プロセスの upid と等しい場合に復元対象としないのは ckpt の選定を行っている dmtcp_restart プロセスが復元処理後に MPI 管理プロセスに変化するからである．もし復元対象とした場合は MPI 管理プロセスを 2 つ復元することになる．次の条件分岐では子プロセスであるかの判定を行う．子プロセスならば MPI 管理プロセスと同じ sid を持ち，childTable 変数に登録済みであるため復元する．

次に本研究において変更を加えた場合の復元対象 ckpt の選定条件について述べる．3.2.2 節で述べたように，一般的な DMTCP は親子関係にないプロセスの ckpt を復元しない．これは図 3.11 の複製条件を見ると明らかである．本研究ではノードが異なれば sid も異なる点に注目する．ノードが異なれば sid も異なることは図 3.10 により明らかである．つまり図 3.11 中の 2 つ目の条件分岐において sid が異なった場合は他ノードの ckpt であることを示す．そこで本研究では target.sid が MPI 管理プロセスと異なる場合でも復元対象とするように条件を加えた．この条件は図 3.11 における 1 つ目の条件分岐の前に加えた．変更を加えた場合の復元対象 ckpt の選定フローチャートを図 3.12 に示す．

図 3.12 に示す処理フローにより，復元対象プロセスの生成が可能となり，次のプロセス間通信の再構築フェーズに移行する．

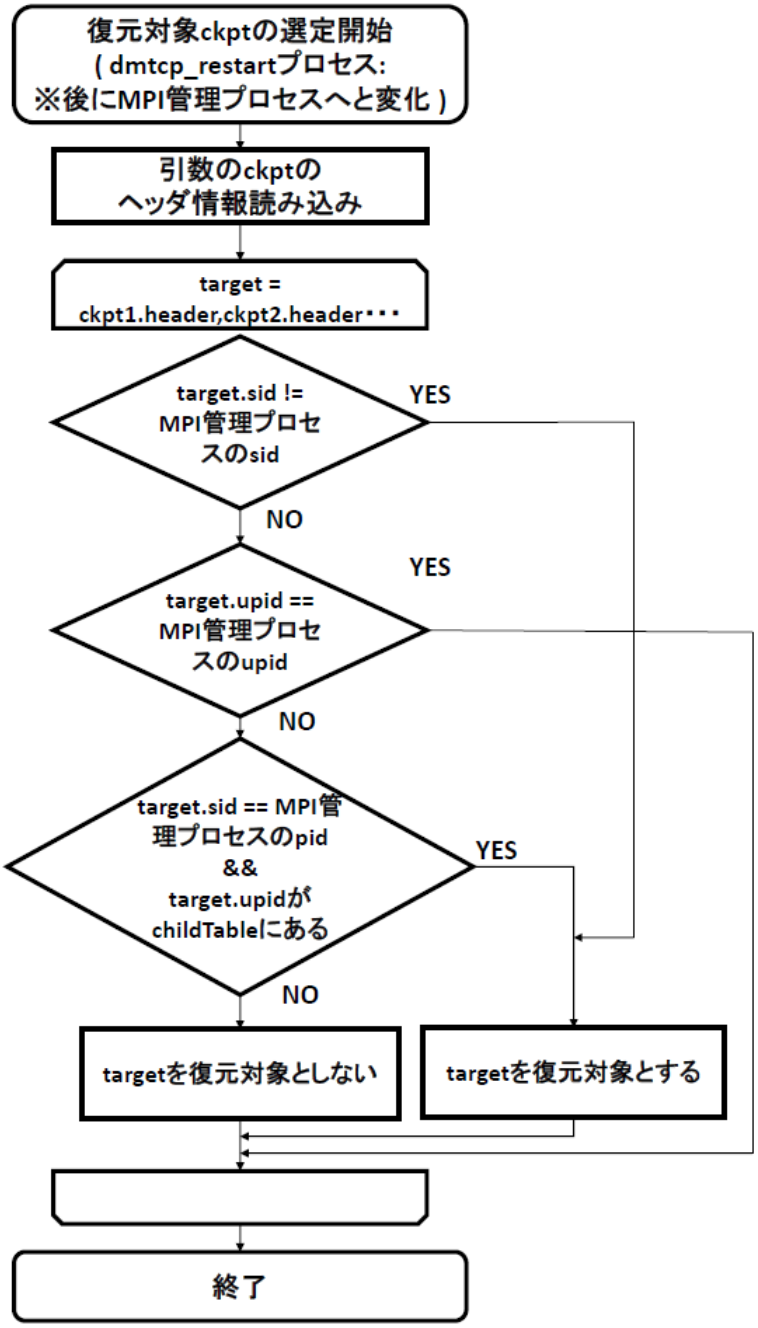

図 3.12　DMTCP 編集後の復元対象 ckpt の選定フローチャート

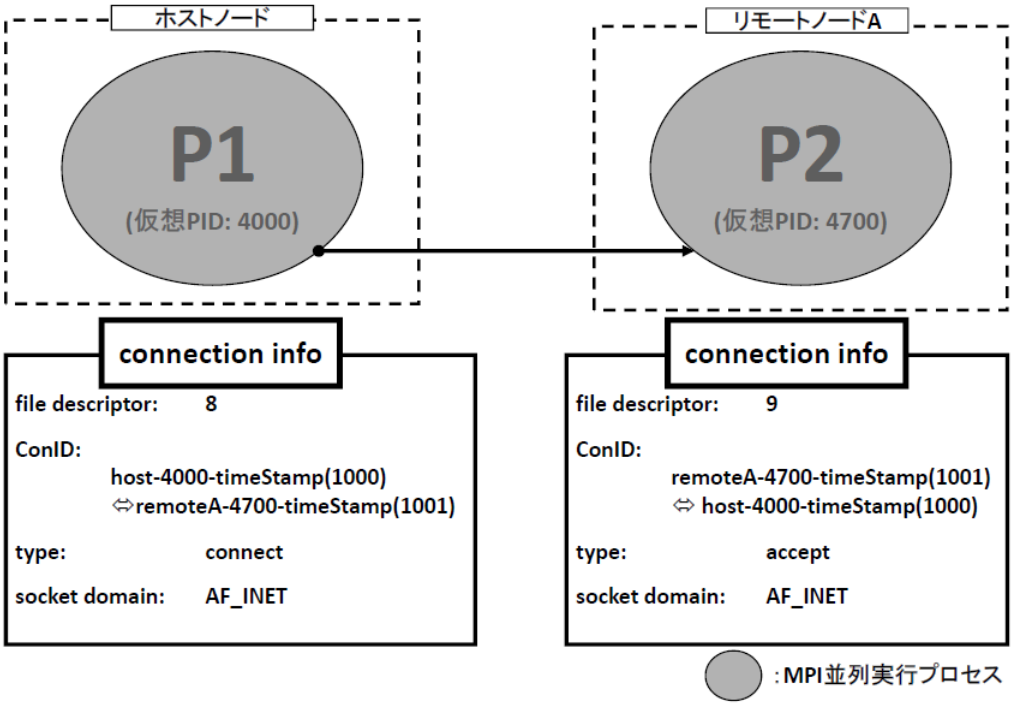

図 3.13　チェックポイント処理時に記録するプロセス間通信に関する情報

3.2.3　一般的なプロセス間通信の再構築と実装上の要件

2.2.1 節で述べたように，DMTCP はプロセス間通信に関する情報を記録している．例えばクラスタ内のどのプロセスと通信するか，または通信方法も記録する．ある 2 つのプロセス間通信を例に，チェックポイント処理時に記録するプロセス間通信に関する情報 (以降　‘connection info’) を図 3.13 に示す.

図 3.13 は MPI 並列実行プロセス P1 と P2 が通信中にチェックポイント処理により記録した connection info である．P1 はソケットファイルディスクリプタ (8) により P2 に接続要求したクライアント側である．一方 P2 はこの要求をソケットファイルディスクリプタ (9) により接続を受け付けたサーバ側である．ここで接続要求は connect() システムコール，接続の受け付けは accept() システムコールを指し，DMTCP がラッパーによりこれらシステムコールを監視/記録しているため，判別可能である．加えて図 3.13 に示すように，プロセス間通信においてクライアント側とサーバ側双方では各通信を識別するための connection ID(以降 ConID) を持つ．ConID の名前規則は ‘ホスト ID-仮想 PID-タイムスタンプ (コネクション番号)’ であり，内部に含むホスト ID や仮想 PID，タイムスタンプなどのデータを表現する．この ConID は 1 対 1 対応しており，クライアント側/サーバ側は対応関係にある通信先の ConID を把握している．この ConID により，各プロセスは通信の再構築を行う通信はいくつあり，またどのプロセスと通信するのか把握することができる．図 3.13 の例では P1 は ConID

である host-40000-timeStamp(1000) に対応する ConID は remoteA-4700-timeStamp(1001) であると把握している．P2 側も同様である．またソケットドメインのタイプも記録し，図 3.13 では TCP を用いたソケット通信であるため，ソケットドメインは AF_INET が情報として保存される．このような conneciotn info はプロセス間通信の数分だけ持つ．例えば図 3.13 中の P1 が他にも 2 つのプロセスと通信中だった場合は connection info を計 3 つ持つ．DMTCP のプロセス間通信の再構築はこの connection info を元に行う．DMTCP におけるプロセス間通信の再構築の流れを以下に示す.

STEP0. 接続要求の受け付け準備:

STEP1. ソケットの reopen:

ckpt 内の記録内容を元にソケットを再作成

STEP2. ConID とアドレス情報の登録:

各プロセスは自身の持つ ConID とアドレス情報を dmtcp_coordinator に登録

STEP3. 接続先プロセスのアドレス情報の問い合わせ:

各プロセスは接続先プロセスのアドレス情報を dmtcp_coordinator に問い合わせて取得

STEP4. 再接続:

問い合わせで取得した情報をもとにコネクションを再確立

STEP5. カーネルバッファの補填:

チェックポイント処理時にサーバ側プロセスのメモリにフラッシュされたネットワーク上のデータを送信側プロセスに返送加えて，クライアント側プロセスはそのデータを再送

STEP0. ではこれからプロセス間通信の再構築を行うにあたり，接続要求を受け付け通信を確立するためのソケットを用意する．DMTCP は TCP を用いた通信と UNIX ドメインソケットを用いた通信両方の通信を再構築するために，両方とも再構築用のソケットを作成し，接続要求を待つ．STEP1. では connection info 中の socket domain タイプに応じてソケットを作成する．次の STEP2. STEP5. までの間は同期を取りながら処理を実行する．同期は dmtcp_coordinator と各プロセスとの間のメッセージ交換により実現する．例えば STEP2. 開始時にはこれから ConID とアドレス情報の登録フェーズに突入することを意味するメッセージを dmtcp_coordinator が各プロセスに送信する．ConID とアドレス情報の登録が完了したプロセスは次の STEP の開始を意味するメッセージの受信待ち状態へ移行する．STEP2. では各プロセスが STEP0. で再構築用に用意した接続要求の受け付け可能なソケット構造体 (以降アドレス情報) と，サーバ側 (accept() システムコールを実行した) プロセスは受信する通信における各 ConID を全て dmtcp_coordinator に登録する．図 3.13 の例ではサーバ側である P2 は

ConID である remoteA-4700-timeStamp(1001) を登録する．登録時は自身の upid や登録データサイズを含んだメッセージを一度送信してから dmtcp_coordinator に登録データを送信する．図 3.13 を例とすると，サーバ側プロセスである P2 は自身のアドレスと ConID である remoteA-4700-timeStamp(1001) を dmtcp_coordinator に登録する．STEP3. では各プロセスは接続先プロセスのアドレス情報を取得するために，クライアント側 (connect() システムコールを実行した) プロセスは送信する通信における各 ConID に対応する受信側の ConID をキーとして dmtcp_coordinator に問い合わせる．STEP2. 同様一度自身の upid や登録データサイズを含んだメッセージを一度送信してから dmtcp_coordinator に問い合わせる．そして dmtcp_coordinator はキーをもとに登録内容を参照し，ヒットした場合は対応したアドレス情報を返信する．図 3.13 を例とすると，クライアント側プロセスである P1 は ConID である host-40000-timeStamp(1000) と 1 対 1 対応している remoteA-4700-timeStamp(1001) をキーとして dmtcp_coordinator に問い合わせる．そして dmtcp_coordinator は P1 に対して P2 のアドレス情報を返信する．STEP4. では STEP3. で取得したアドレス情報をもとにプロセス間通信の再接続を行う．STEP4. の処理のフローチャートを図 3.14 に示す．

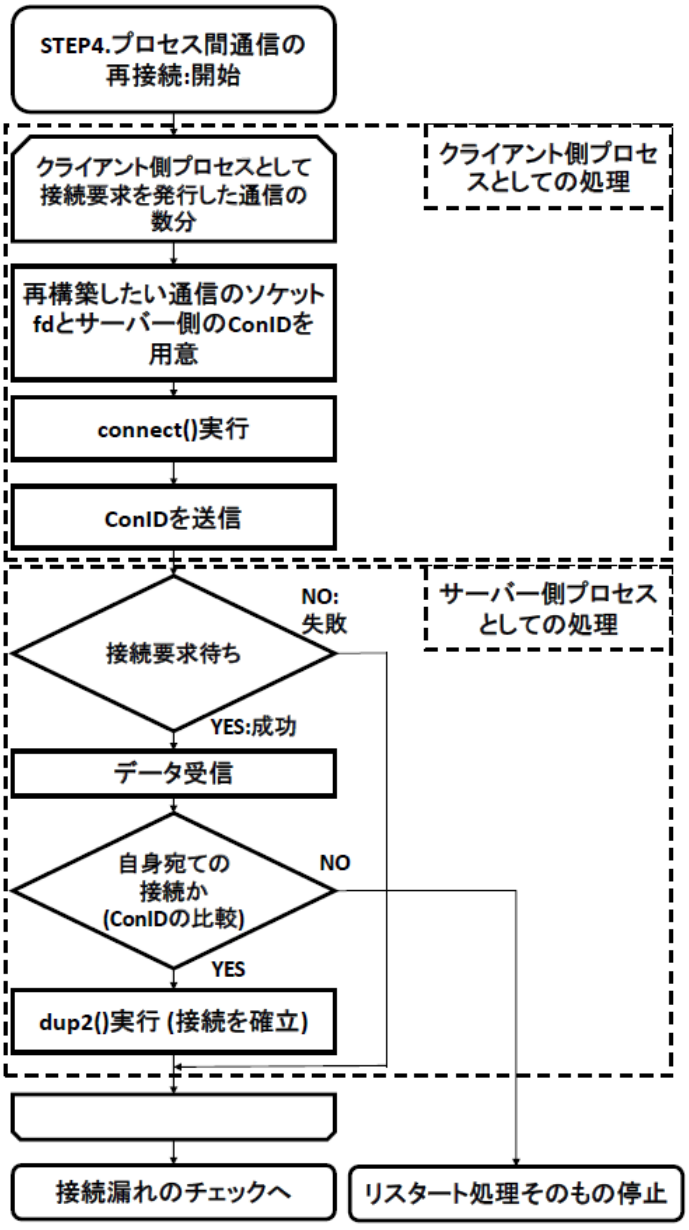

図 3.14　STEP4 における接続確立までのフローチャート

図 3.14 に示すように各プロセスはクライアント側プロセスとしての処理とサーバ側プロセスとしての処理を行う．まずクライアント側プロセスとして，connection info 内に記録したソケット用 fd でサーバ側に connect() システムコールを実行し，接続を要求する．同時に 1 対 1 対応しているサーバ側の ConID を送信する．次にサーバ側プロセスとして accept() システムコールを実行し接続要求を待つ．接続要求を受け付けた後，クライアント側プロセスが送信した ConID を読み込む．各プロセスは自身がサーバ側である通信を ConID により把握しているため，読み込んだ ConID が自身宛の通信であるか確認する．もし自身宛の接続ならば，connection info 中に記録している fd を指定して dup2() システムコールを実行する．dup2() システムコールにより接続を受け付けたソケットの fd が connection info 中に記録している fd に置き換わり，チェックポイント処理時と同じソケット fd を使用したソケット通信を再現できる．もし自身宛の接続でなければ assert 機能によりリスタート処理そのものを停止します．図 3.13 を例とすると，P2 は accept() システムコールを実行し接続要求を待つ．P1 は connect() システムコールを実行し，接続を要求する．加えてサーバ側の ConID である remoteA-4700-timeStamp(1001) を送信する．P2 は P1 からの接続要求を受け付け，加えてデータとして remoteA-4700-timeStamp(1001) を受信する．P2 は自身がサーバ側である通信は remoteA-4700-timeStamp(1001) ということを把握しているため，受信データの remoteA-4700-timeStamp(1001) は再構築すべき通信の ConID であると判断する．そして dup2() システムコールを実行し，接続を受け付けたソケットを connection info 内の fd(9) に置き換え，他の接続要求に備える．しかしタイミングによっては接続漏れが発生する通信がある．DMTCP はタイミングによって接続を確立し損じた接続要求も受け付けるフォロー機能も持つ．各プロセスは通信の再構築を行う通信の数を記録しているため，接続を確立した通信の数が記録内容よりも少ない場合はブロッキングモードで accept() システムコールを実行し，確実に接続要求を受け付けるフェーズを設けている．このフォロー機能のフローチャートを図 3.15 に示す．

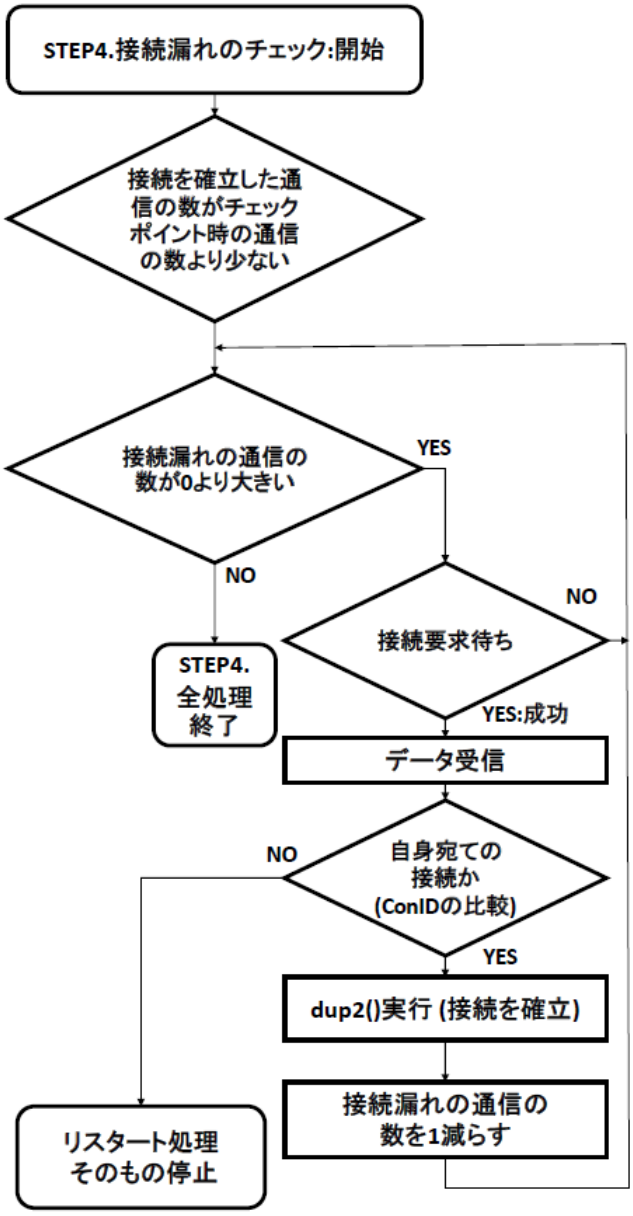

図 3.15　STEP4 におけるブロッキングモードによる再接続のフローチャート

図 3.15 において，ブロッキングモードで接続待ちを行っているため，本来接続要求の受け付けには失敗しない．このようにブロッキングモード機能を備えているため接続漏れの発生を防ぎ，チェックポイント処理時のプロセス間通信を完璧に再構築する．

最後に STEP5. ではチェックポイント処理時にカーネルバッファ内のネットワーク上のデータは受信側プロセスのメモリにフラッシュされるため，このネットワークデータを元の送信側プロセスに返送し，送信側プロセスはデータを再送信してカーネルソケットバッファを再充填する．以上が一般的な DMTCP におけるプロセス間通信の再構築の流れである．

次にプロセス単位での再配置を実現する上で，プロセス間通信の再構築時における課題を述べる．その課題とは，プロセス単位での再配置時において，チェックポイント処理時に記録したプロセスの一部が復元されず，プロセス間通信の再構築が完全に行われないことである．具体例を図 3.16 に示す．

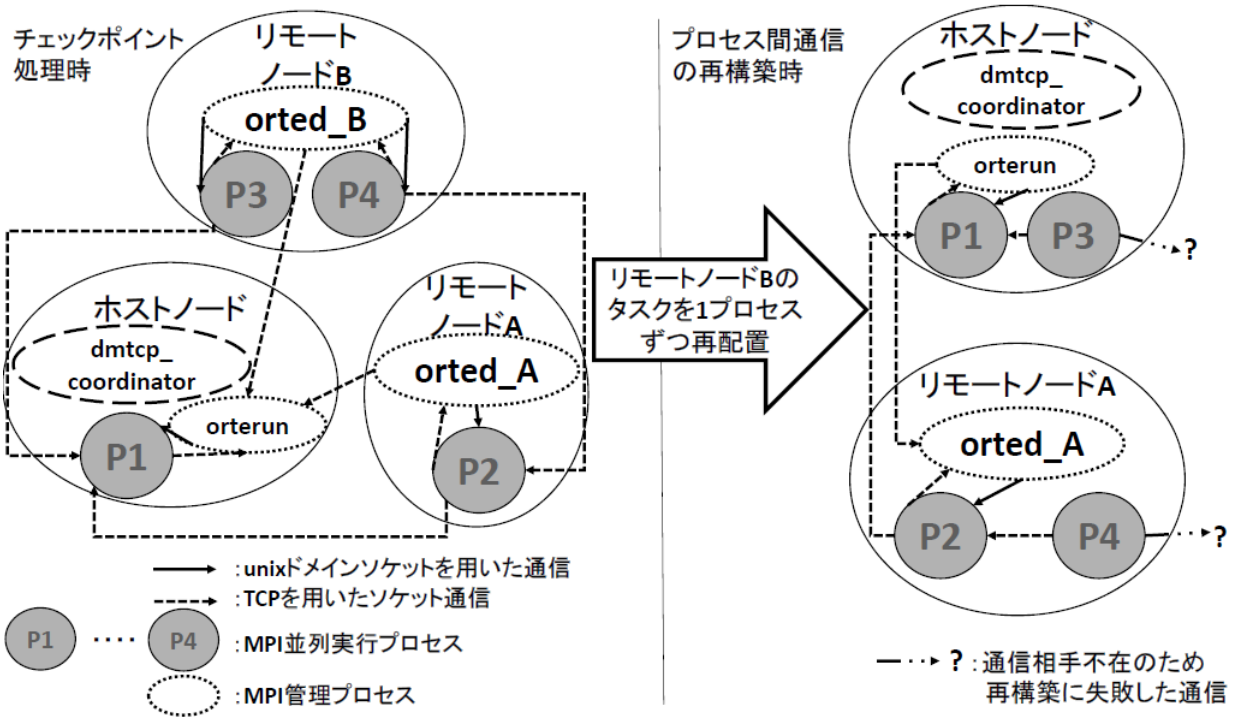

図 3.16　プロセス間通信の再構築の失敗例

図 3.16 の左側の図はチェックポイント処理時に記録したクラスタであり，3 ノードで構成され，そのうちリモートノード B では 2 プロセス起動しているクラスタである．右側の図は左側のクラスタからリモートノード B が脱退したと仮定し，ホストノードとリモートノード A それぞれに 1 プロセスずつ再配置した際のプロセス間通信の再構築を示している．

3.1 節中の図 3.2 で示したように，前提条件としてプロセス間通信は全てネットワークを介した通信に設定するため，図 3.16 中のプロセス間通信は全て TCP を用いたソケット通信である．通信方法がネットワークを介した通信ならばプロセスがクラスタ内のどのノードに移動していも通信は再構築できる．しかし ‘脱退したノード内で起動した orted は復元しない’ というもう 1 つの前提条件により，図 3.16 中でプロセス間通信の再構築に失敗する通信が発生する．図 3.16 中で脱退したノード内の orted_B は復元しないため，orted_B と通信していた P3，P4 間における通信の再構築に失敗する．加えて orterun 側においても orted_B からの接続要求がないため，接続漏れと判断しブロッキングモードに移行して常に接続要求を待ち続ける．そのため STEP5. に移行できない問題もある．そこでプロセス間通信の再構築時において，チェックポイント処理時と通信相手が異なる場合も通信の再構築を可能とする必要がある．

3.2.4 通信形態の動的変更に対応した通信の再構築

チェックポイント処理時と通信相手が異なる場合も通信の再構築を可能とするための要件を検討する．要件は以下の 3 つである．

要件 1. 復元しないプロセスへの通信を別のプロセスへ変更する

要件 2. チェックポイント処理時に記録した通信以外の接続要求を受け付け可能とする

要件 3. 復元しないプロセスとの間の通信は再構築しない

要件 4. 要件 2. で実現した新しい通信においてもブロッキングモードを活用して接続漏れがないようにする

要件 5. 再構築しない通信においてはカーネルバッファの補填 (STEP5.) は行わない

本研究では上記の要件を満たすために DMTCP に変更を加え，チェックポイント処理時と通信相手が異なる場合も通信の再構築を可能とする．本節は各要件を満たすための実装方法についてそれぞれ述べる．

要件 1. は復元しない orted とその子プロセスである MPI 並列実行プロセスとの間の通信を再構築時に変更する．プロセス間通信の再接続のために再配置されるプロセスは orted のアドレス情報を問い合わせるが接続可能なアドレスを取得できない．復元されない orted は自身へ接続するためのアドレス情報を登録していないため当然の結果である．本研究では dmtcp_coordinator がアドレス情報取得の問い合わせに対して，アドレス情報を発見できなかった場合は問い合わせ元のノード (IP アドレス) を手がかりにその IP アドレス内で起動している MPI 管理プロセスへ接続できるアドレス情報を代わりに返信するよう処理を追加する．そのためには各ノードで起動している MPI 管理プロセスと MPI 管理プロセスへ接続可能なアドレス情報を把握する必要があり，本研究ではこの 2 つの処理を dmtcp_coordinator に追加した．新しい処理を加えた dmtcp_coordinator が MPI 管理プロセスを把握し，接続先プロセスのアドレス情報の問い合わせに対して返信するまでのシーケンス図を図 3.17 に示す．

図 3.17 中に登場するオブジェクトは 4 つあり，左から dmtcp_coordinator，復元後親プロセスである MPI 管理プロセス (orterun もしくは orted) に変化するプロセス，MPI 管理プロセスが一緒に復元されかつ復元後は MPI 並列実行プロセスに変化するプロセス，MPI 管理プロセスが一緒に復元されずかつ復元後は MPI 並列実行プロセスに変化するプロセスである．最後に列挙したプロセスは別の表現をするとプロセス単位で再配置するプロセスである．図 3.17 において，復元対象プロセスの生成後に各プロセスは DMTCP の管理下にいることを示すために自身の upid を含んだメッセージを dmtcp_coordinator に送信する．この時各ノードから送られてくるメッセージの 1 番最初が MPI 管理

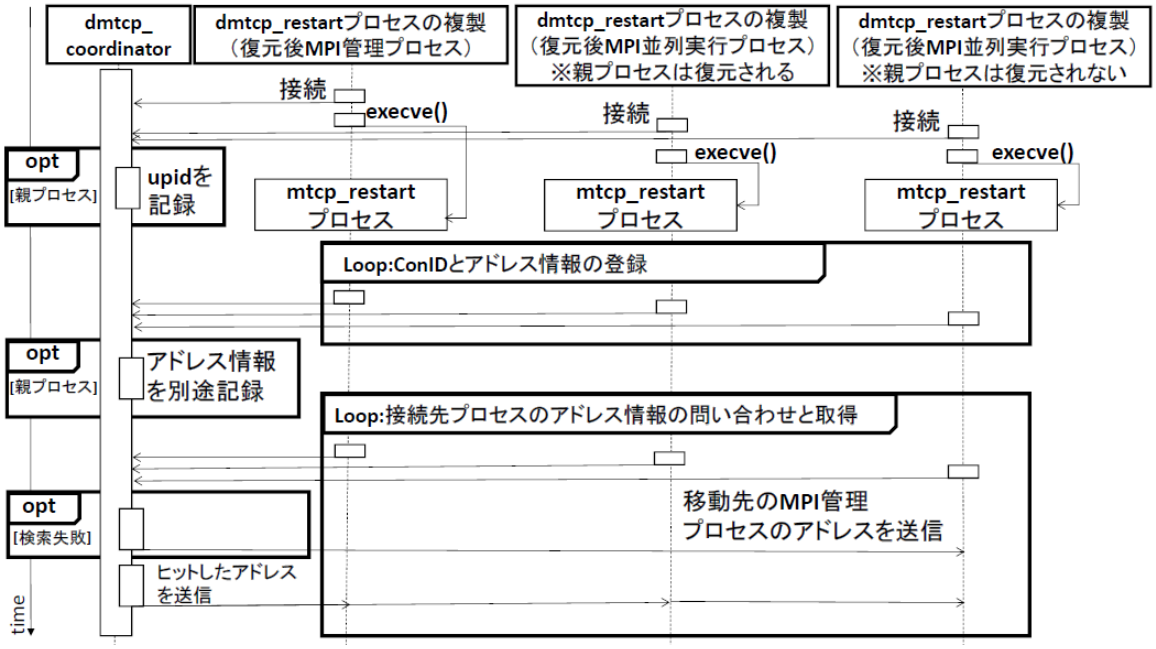

図 3.17　プロセス生成から通信先のアドレス情報を取得するまでのシーケンス

プロセスであることを活用し，MPI 管理プロセスの upid を専用の配列に別途記録する．次に ConID とアドレス情報の登録フェーズ時，登録の直前に送られているメッセージをもとにどのプロセスから登録依頼を受けたか確認する．確認にはメッセージ内の upid を使用し，MPI 管理プロセスの upid を格納した配列と比較する．もし MPI 管理プロセスからの登録依頼ならば通常の登録処理に加えて，そのアドレス情報を参照できるポインタを別途用意し記録する．記録時にはその MPI 管理プロセスが起動している IP アドレスを対応させて記録し，IP アドレスがキーとなる．次の接続先プロセスのアドレス情報の問い合わせフェーズでは問い合わせに対して ConID をキーにアドレス情報を検索する．この時，図 3.17 の一番右のプロセスは復元しない MPI 管理プロセスのアドレス情報は取得できないため，取得に失敗する．当然だがこの復元しない MPI 管理プロセス以外のプロセスのアドレス情報は検索に成功し，アドレス情報は取得できる．もしアドレス情報の検索に失敗した場合はその問い合わせもとの IP アドレスをキーに，その IP アドレスが指すノード内で起動している MPI 管理プロセスのアドレス情報を検索結果として送信する．これにより再配置されるプロセスが発行する接続要求先を再配置先ノード内の MPI 管理プロセスへ変更する．

要件 1. により，再配置先ノード内の MPI 管理プロセスはチェックポイント処理時には記録していない通信の接続要求を受けることになる．DMTCP は図 3.14 に示すように自身が復元すべき接続か判断し，違う場合はリスタート処理そのものを停止する．そのため要件 2. はチェックポイント処理時に記録した通信以外の接続要求を許可し，以降の再構築処理を継続可能とする．実装方法はチェックポイント処理時に記録した通信以外の接続要求を受けた場合は新しく受信用ソケットを作成し，そ

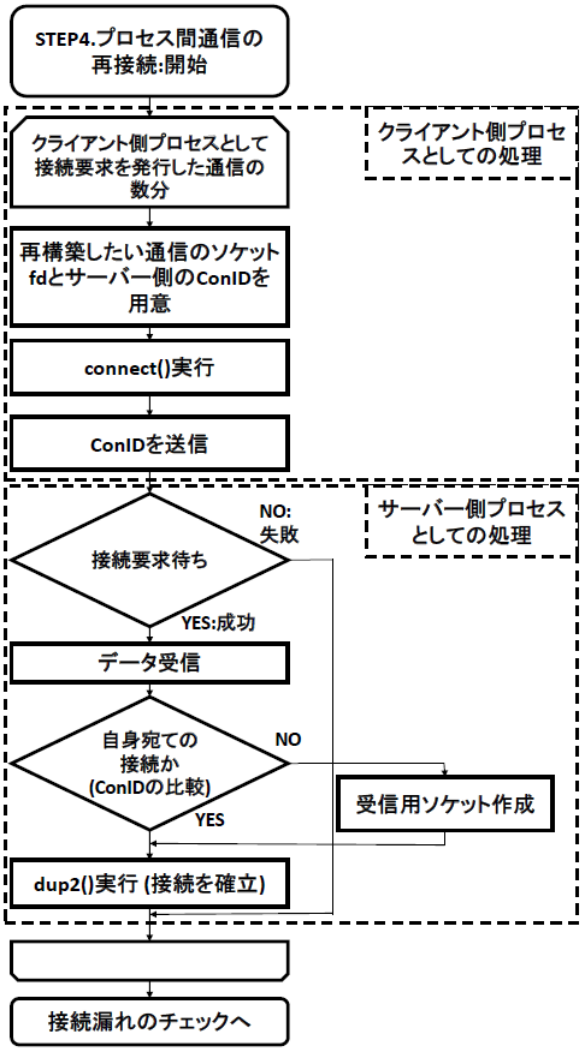

図 3.18　接続確立までのフローチャート (記録にない接続要求も受け付け可能)

のソケット fd に対して dup2() システムコールを実行し，通信を確立するというシンプルな方法である．図 3.18 に上記の実装方法を加えた場合における再接続処理のフローチャートを示す.

図 3.18 において，一般的な DMTCP であれば読み込んだ ConID が自身宛ての ConID でなければ assert 機能によりリスタート処理そのものが停止する (図 3.14) が，新しく受信用ソケットを用意し，接続を確立する．ただし作成するソケットには制約を設ける．DMTCP はソケット fd も記録し復元するため，作成したソケットが復元予定のソケットと重複してはいけない．図 3.18 中で接続確立用のソケットソケット作成時には復元予約済みのソケット fd と比較し，重複しない fd を受信用ソケット fd に採用する．

要件 3. では復元しないプロセスとの間の通信を再接続しないと判断することで次の STEP(カーネルバッファの補填) に進むことができる．再接続しないと判断しなければならない通信パターンは 2 つである．

通信パターン 1. 復元しない orted とその子プロセスである再配置するプロセスとの間の通信

通信パターン 2. orterun と復元しない orted 間の通信

上記の 2 パターンの通信に関しては再接続しないと判断しなければ接続要求を常に待ち続け次の STEP に移行できない．1 つ目の通信パターンにおいては，‘UNIX ドメインソケットを用いて通信するのは親プロセスとの間の通信のみである ‘という特徴を活用する．再配置するプロセスに限定して，UNIX ドメインソケットを用いた通信の接続漏れは再接続しないとする．再配置するプロセス以外は通常通り UNIX ドメインソケットを用いた通信の接続漏れについてはブロッキングモードでの接続要求待ち状態へと移行し，接続を保証する．2 つ目の通信パターンにおいては orterun がどのノードが脱退したかを把握することで再接続しないと判断できる．把握する方法にはホスト ID を活用する．このホスト ID はノードごとに割り当てる固有の ID であり，ConID 内にも含まれている．本研究ではチェックポイント処理時に記録した通信相手のホスト ID とリスタート処理時にクラスタ内に残っているノードのホスト ID を比較し，どのノードが脱退し orted からの接続要求を待つ必要がないのか判断する．複数の orted と通信する orterun に対してチェックポイント処理を行った場合に記録する ConID の例を図 3.19 に示す.

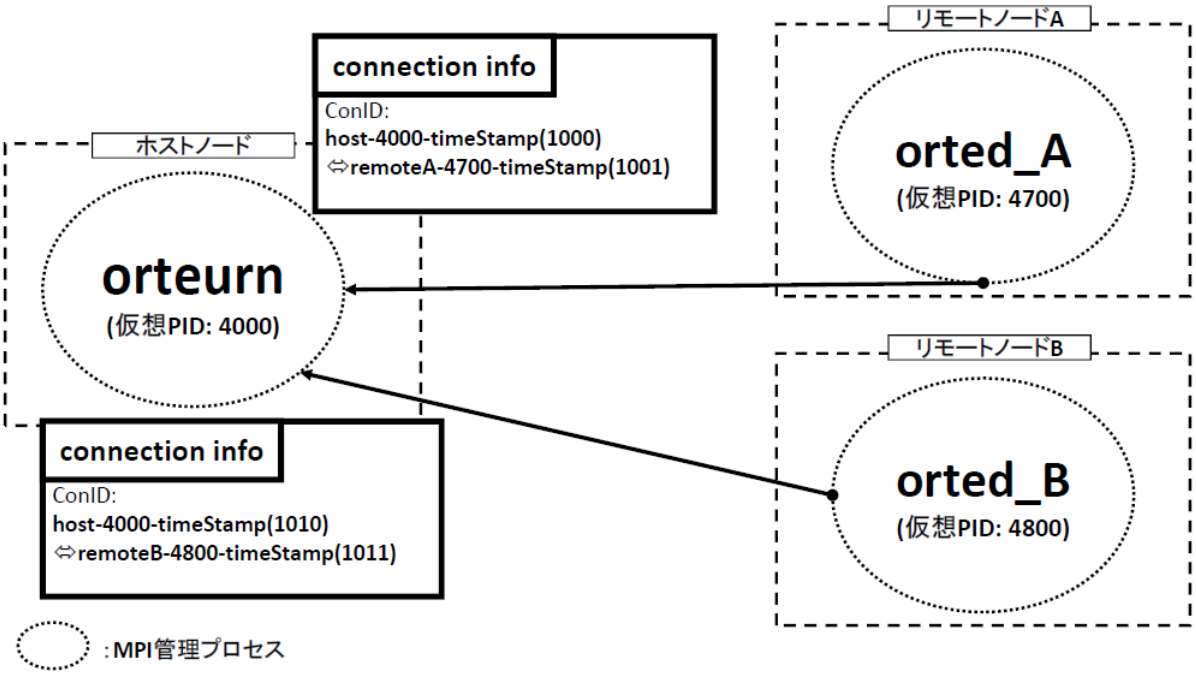

図 3.19　複数の orted と通信する orterun において記録する ConID 例

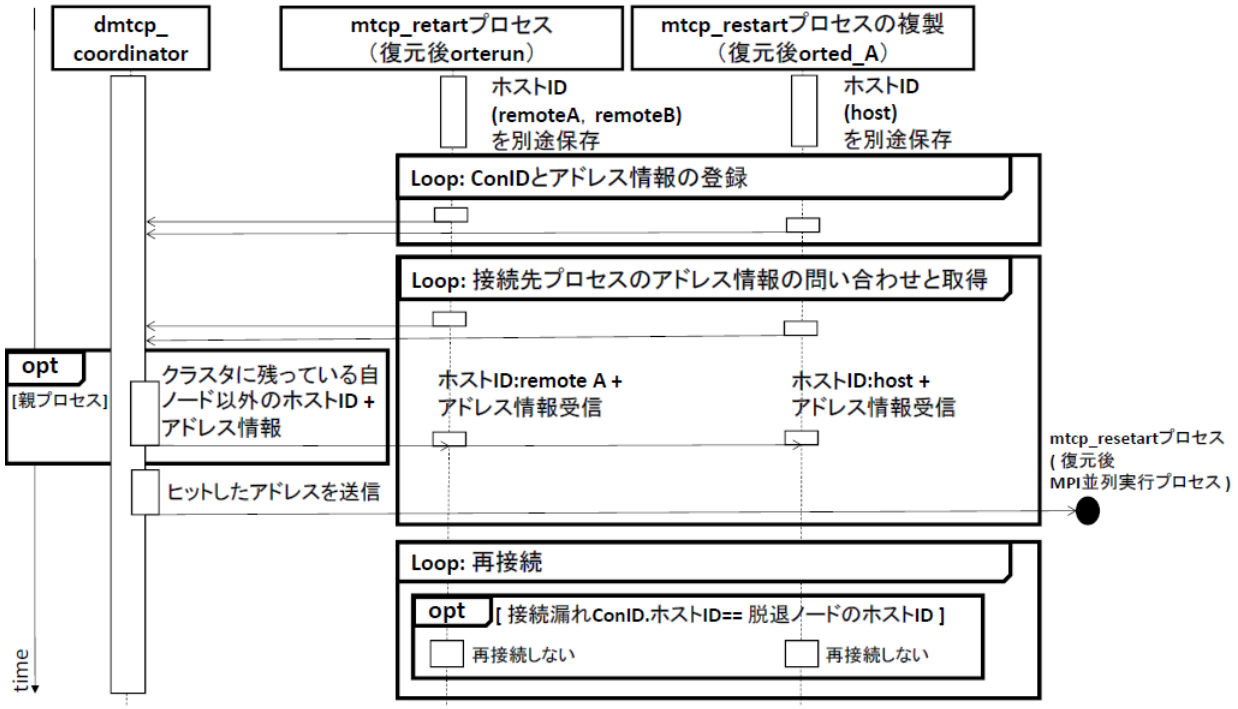

図 3.20　図 3.19 におけるリモートノード B 脱退時のプロセス間通信の再構築シーケンス

図 3.19 ではチェックポイント処理時に orterun は 2 つの orted と通信し，リスタート処理時には図 3.19 中の connection info を活用してプロセス間通信の再構築を始める．本研究ではプロセス間通信の再構築時においてアドレス情報の登録フェーズ前に，記録した connection info の中から通信相手がリモートノードである ConID からホスト ID を抽出して別途保存する．図 3.19 を例にすると，ホスト ID を示す remoteA，remoteB の 2 つのホスト ID を保存する．そしてアドレス情報の問い合わせフェーズ時に dmtcp_coordinator から MPI 管理プロセスに対してクラスタ内に残っているノードのホスト ID を通知する．通知方法は問い合わせメッセージに付加して送信することとする．図 3.19 において，リスタート処理時にリモートノード B が脱退した場合に orterun に orted_B が存在しないことを通知し，再接続フェーズを終えるまでのシーケンス図を図 3.20 に示す．

図 3.20 において，親プロセスである orterun と orted_A は他ノードとの通信を行う ConID からそのホスト ID を抽出し別途保存する．orteru は remoteA，remoteB の 2 つのホスト ID を保存する．orted_A はホストノードのホスト ID である host を保存する．次にアドレス情報の問い合わせフェーズ時に orterun，orted_A は dmtcp_coordinator からクラスタ内に残っているノードのホスト ID を受け取る．最初に別途保存したホスト ID と dmtcp_coordinator から受け取ったホスト ID を比較することでホスト ID が remoteB であるリモートノード B が脱退したことを把握できる．そして次の再接続フェーズにおいて，接続漏れの ConID が持つホスト ID が脱退したノードのホスト ID であればブロッキン

グモードに移行して接続要求を待つのではなく，再接続しない通信として判断する．以上のようにどの通信を復元しないかを把握することで接続要求を待ち続けることなく，次の接続要求待ちあるいは次の処理に進むことができる．復元しない通信に使用するソケット fd は閉じる．

要件 4. では要件 2. の実現によって発生する新しい通信について接続漏れがないことを保証する．チェックポイント処理時に記録した通信は接続漏れを検知できるが，通信の再構築時に発生した新しい通信は記録にないため接続漏れを検知できない．要件 4. を満たすことでこの記録にない通信の接続漏れを検知し，ブロッキングモードに移行し確実に接続する．本研究では各ノードに新たに再配置するプロセスの数を活用して，新しく発生する通信を把握する．チェックポイント処理時に記録した通信以外の通信が発生する場合は，再配置するプロセスと復元しない orted との間の通信を要件 2. により復元しない orted ではなく再配置先の MPI 管理プロセスへ変更した場合だけである．そのため新しく発生する通信の数は再配置したプロセスの数に等しい．この再配置したプロセスの数は再配置先ノードで起動している総プロセス数とチェックポイント処理時に記録した子プロセスの数との差分で得ることができる．チェックポイント処理時に記録した子プロセスの数は childTable のサイズにより得ることができる．しかしリスタート処理時にノードで起動している総プロセス数の情報は親プロセスである MPI 管理プロセスは保持していない．そこで dmtcp_coordinator から親プロセスに対して，各ノードで起動している総プロセ数を通知する．再配置したプロセスの数をもとに，プロセス間通信の再構築を行うシーケンス図を図 3.21 に示す．

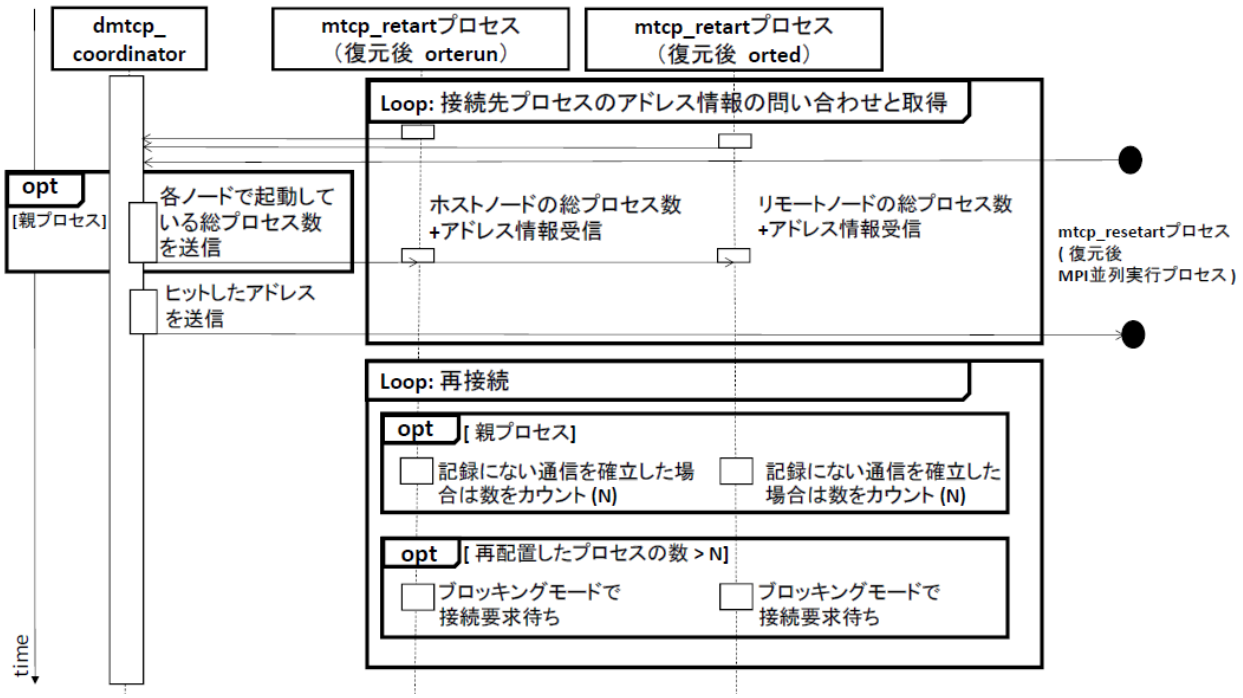

図 3.21　新しい通信の発生に対応したプロセス間通信の再構築シーケンス

図 3.21 では，再配置したプロセスの数を活用した制御を加えたプロセスである親プロセス (orteurn と orted) を要素として登場させている．まずアドレス情報の問い合わせフェーズ時に問い合わせ元ノードで起動しているプロセスの総数をメッセージに付加して送信する．この時点で，総プロセス数から childTable のサイズを引くことで再配置したプロセス，つまり新しく発生する通信の数を把握できる．そして新しく発生する通信の数を把握した上で，再接続フェーズに移行する．再接続時，チェックポイント処理時に記録した通信以外の通信を確立した場合はその数をカウントする．図 3.21 においてはカウントした数を N とする．このカウントした数 N が再配置したプロセスの数より少ない場合は接続漏れと判断し，ブロッキングモードでの接続要求待ち状態へと移行する．これによりチェックポイント処理時に記録した通信以外の通信も確実に接続を確立することができる．

要件 5. では再構築しない通信 (正確にはソケット) に対してはカーネルバッファの補填は行わないことで，プロセス間通信の再構築が完了し，アプリケーションが再開する．実装方法はシンプルで要件 3. で再接続しないと判断したソケット fd を記録する．カーネルバッファの補填時，記録した fd ついては補填処理を行わない．

3.3 実現した要件のまとめ

本研究では DMTCP における復元対象プロセスの生成，プロセス間通信の再構築に変更を加え，これまで述べてきた要件を全て満たすことでプロセス単位での並列タスクの再配置による負荷分散が実現できる．説明量が多かったため，改めて本章で実現した要件を図 3.22 に示す．

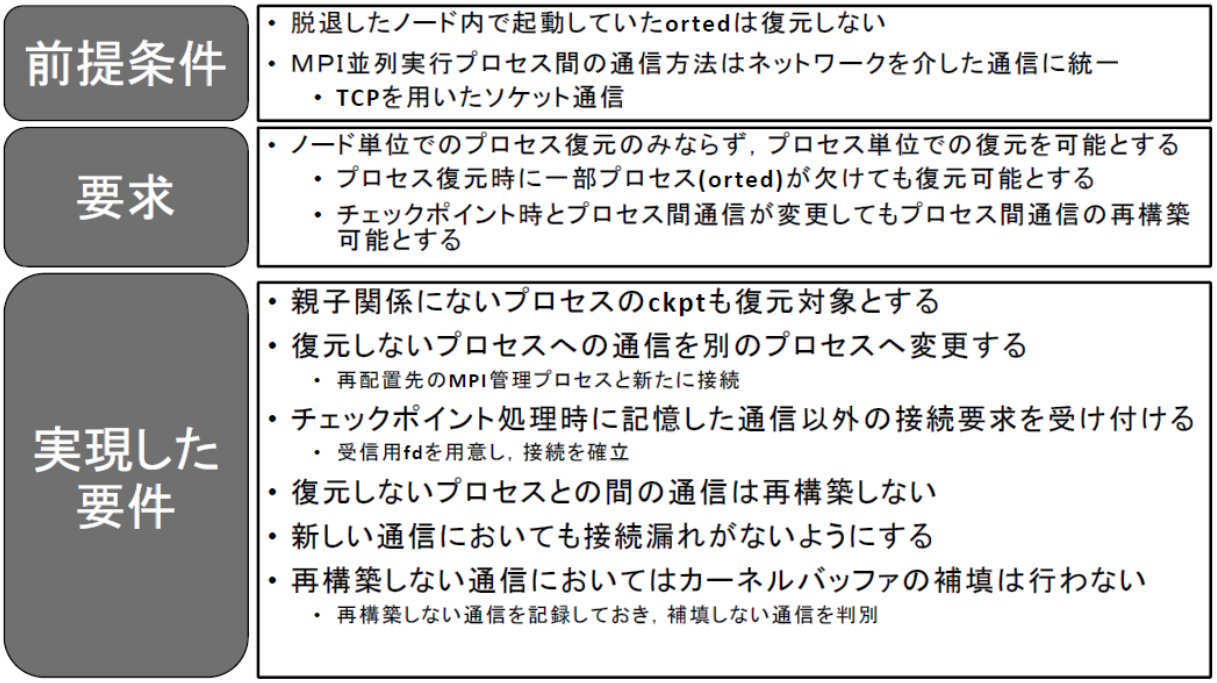

図 3.22　プロセス単位での再配置機能のために実現した要件

第4章　効率的通信の実現

本章はプロセス単位での並列タスクの再配置により，プロセス間通信の方法が非効率な通信が発生する問題を解消ために，DMTCP に追加したプロセス間の効率化機能について述べる．

4.1　プロセス間通信の効率化機能の概要と実装上の要件

2.3 節で述べたように，DMTCP は並列タスクの再配置 (移譲) の有無に関係なく，チェックポイント処理時と同様のプロセス間通信を再構築する．例えばノード間通信を行っていたプロセス同士が並列タスクの再配置により同一ノードで起動した場合でもノード内通信ではなくノード間通信で通信が再構築される．そこでリスタート処理時，通信相手先プロセスが同一ノード内であるかを判断材料にプロセス間通信の方法を切り替える処理を加える．この処理ではチェックポイント処理時には TCP を用いたソケット通信を行っていたプロセス同士がリスタート処理時に同一ノード内で起動した場合は高速なノード内通信に切り替える．ここで切り替える通信方法について検討する．本研究ではプロセス単位で並列タスクを分散して複数ノードに再配置するために，前提条件として MPI 並列実行プロセス間の通信方法は TCP を用いた通信に限定した．MPI が提供する通信方法の中でより高速な通信方法として共有メモリ通信が挙げられる．しかし mtcp_restart プロセスとして復元処理中に共有メモリ通信に変更するにはプロセス間通信の再構築が複雑になると予想する．もしくは mtcp_restart プロセスとして復元処理中に共有メモリ通信に変更することは不可能であり，プロセス復元後に MPI 側で動的に変更しなければならない可能性もある．そこで本研究では UNIX ドメインソケットを用いた通信に注目する．UNIX ドメインソケットを用いた通信はノード内通信の方法であり，TCP を用いたソケット通信より高速である．また 3 で示したように，DMTCP はソケット通信のうち TCP を用いたソケット通信と UNIX ドメインソケットを用いた通信を区別して再構築している点から DMTCP 側で通信方法を変更できるのではないかと考えた．そこで TCP を用いたソケット通信と UNIX ドメインソケットを用いた通信の性能を比較する．評価方法は 1 ノード内で親プロセスと子プロセス 2 つのプロセスを起動し，両プロセス間でそれぞれの通信方法でデータを送受信する．評価環境を表 4.1 に示す．そして，TCP を用いたソケット通信を行う場合と UNIX ドメインソケットを用いた通信を行う場合における転送レート (単位:Mbit/second) をそれぞれ示す．また評価

表 4.1　評価環境

CPU	Core i7 4770 3.40GHz
Memory	32GB
OS	Linux (CentOS 7)

表 4.2　(送受信する総データサイズ:10^8Byte, バッファサイズ:1000Byte) 時の転送レート

通信方法	転送レート:min(Mb/s)	転送レート:max(Mb/s)	転送レート:ave(Mb/s)
UDS	11654	13797	12701.2
TCP	2955	4162	3924.9

プログラムでは送受信の任意のバッファサイズを指定可能なため，複数のバッファサイズにおける転送レートを表 4.2，4.3，4.4 に示す.

表 4.2，4.3，4.4 はバッファサイズがそれぞれ 1000Byte，5000Byte，10000Byte の場合の転送レートを 10 回計測し，その中で最小値と最大値，平均値を示している．表 4.2，4.3，4.4 より，通信方法が UDS(UNIX Domain Socket) である場合の転送レートの平均値は通信方法が TCP である場合に比べて 2.5 倍以上である．最大でバッファサイズ 1000Byte の場合に UNIX ドメインソケットを用いた通信の転送レートは TCP の転送レートの 3.2 倍である．よって，UNIX ドメインソケットを用いた通信の通信性能は TCP を用いた通信の通信性能よりも高いことが分かる．加えて送受信のバッファサイズが小さいほど通信性能の差が開くことも同時に分かる．そこで本研究ではリスタート処理時に，同一ノード内で通信するプロセスのプロセス間通信の方法を高速な UNIX ドメインソケットを用いた通信へ変更する機能を加える．そのためには各プロセスが通信相手先プロセスの起動場所 (IP アドレス) を把握し，もし通信方法を切り替える場合は再接続を開始する前に UNIX ドメインで作成したソケットを用意する必要がある.

表 4.3　(送受信する総データサイズ:10^8Byte, バッファサイズ:5000Byte) 時の転送レート

通信方法	転送レート:min(Mb/s)	転送レート:max(Mb/s)	転送レート:ave(Mb/s)
UDS	38957	42618	41031.5
TCP	8128	18192	16008.1

表 4.4 (送受信する総データサイズ:10^8Byte, バッファサイズ:10000Byte) 時の転送レート

通信方法	転送レート:min(Mb/s)	転送レート:max(Mb/s)	転送レート:ave(Mb/s)
UDS	28745	64987	53499.7
TCP	9660	30257	21320.4

4.2 高速な通信方法への変更

プロセス間通信の効率化機能を実現するために，プロセス間通信の再構築に処理を追加した．改めて DMTCP におけるプロセス間通信の再構築の流れを以下に示す.

STEP0. 接続要求の受け付け準備:

STEP1. ソケットの reopen:

STEP2. ConID とアドレス情報の登録:

STEP3. 接続先プロセスのアドレス情報の問い合わせ:

STEP4. 再接続:

STEP5. カーネルバッファの補填:

4.1 節で検討したプロセス間通信の効率化機能を実現するための要件は以下の 2 つである．

要件 1. 通信相手先プロセスが同一ノード内で起動しているか把握する

要件 2. 再接続フェーズに移行する前に適切なドメインのソケットを用意する

要件 1 . では UNIX ドメインソケットを用いた高速な通信に変更するために，通信相手先プロセスの起動場所を把握する．プロセスの起動場所を把握することで，通信相手先プロセスが同一ノードで起動しているかどうかの判断が可能となる．要件 2. では要件 1. を実現した上で，通信相手先プロセスが同一ノード内で起動している場合には UNIX ドメインソケットを用いた高速な通信に切り替えるために UNIX ドメインでソケットを作り直す．通信相手先プロセスの起動場所を把握するタイミングとソケットを作り直すタイミングについて検討する．DMTCP の一般的なプロセス間通信の再構築の流れを 3.2.3 節で述べたが，‘STEP1. ソケット reopen’ 時にドメインを変更してソケットを作ることはできない．STEP1. の実行時，プロセス自身は他のプロセスがどのノード (IP アドレス) で起動したかを把握していない．加えて mtcp_restart プロセスに変化後，STEP2. (ConID とアドレス情報の登録) を行うための同期メッセージを受信するまで他のプロセスとの間の通信は発生しないた

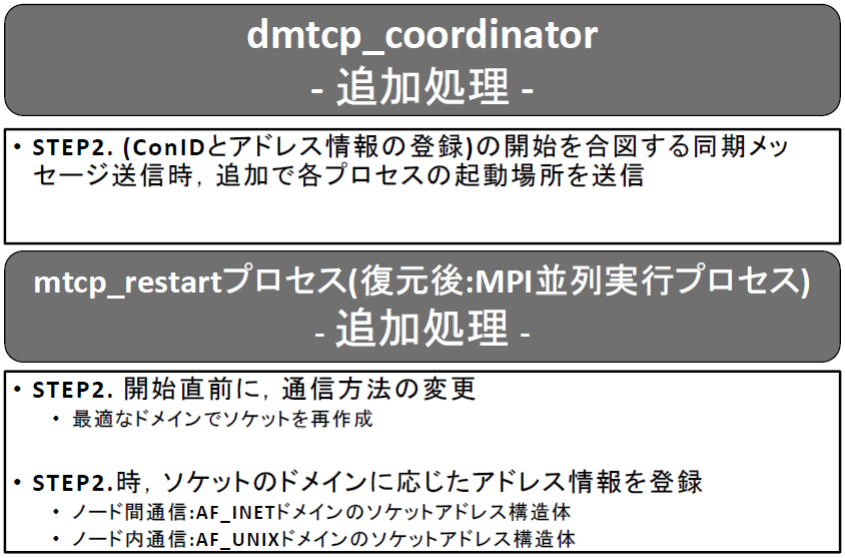

図 4.1　プロセス間通信の効率化機能実現のための検討内容

め，プロセスの起動場所をお互いに教えることもできない．もし STEP2. 以降でプロセスの起動場所に応じて通信方法を変更する場合は dmtcp_cooridinator 側の処理が複雑になる．例えば STEP3. の‘接続先プロセスのアドレス情報の問い合わせ’ 時，dmtcp_coordinator は各問い合わせに対して通信相手先が同一ノード上で起動しているか判断しかつ適切なアドレス情報を検索し，返送する処理フローとなる．よってシンプルな実装とするために，dmtcp_coordinator が STEP2. の開始を合図する同期メッセージを送信する際にプロセスの起動場所を各プロセスに通知することを考える．STEP2. で‘ConID とアドレス情報の登録’ 前に，通信を切り替えるかどうかの判断が行えればソケットのドメインに応じた受信用ソケットアドレス構造体 (アドレス情報) を登録できる．そして STEP3. において dmtcp_coordinator 側でははは追加処理は発生せず，これまで通り各プロセスから送られてきたキー (ConID) をもとに登録内容からアドレス情報を検索し，返送するだけである．これまで検討した内容を図 4.1 にまとめる．

図 4.1 中の mtcp_restart プロセスにおける追加処理において，作り直すソケットの fd はチェックポイント処理時に記録した fd と同じする．あくまで変更するのはソケットのドメインのみである．また STEP2. 中で登録するアドレス情報は STEP0. で各プロセスが再構築用に用意した接続要求を受け付けるソケットアドレス構造体である．

次に通信方法の切り替え判断に用いるデータ (‘各プロセスの起動場所’) について述べる．通信方法の切り替えを判断するに通信相手先プロセスがどのノードで起動していて，記録内容ではどの通信方法で再構築することになっているか把握する必要がある．DMTCP は受信側の通信において，TCP を用いたソケット通信と UNIX ドメインソケットを用いた通信それぞれの ConID を分けてテーブルに保存しているため，どの通信方法で再構築する予定であるかはこのテーブルを参照することで把

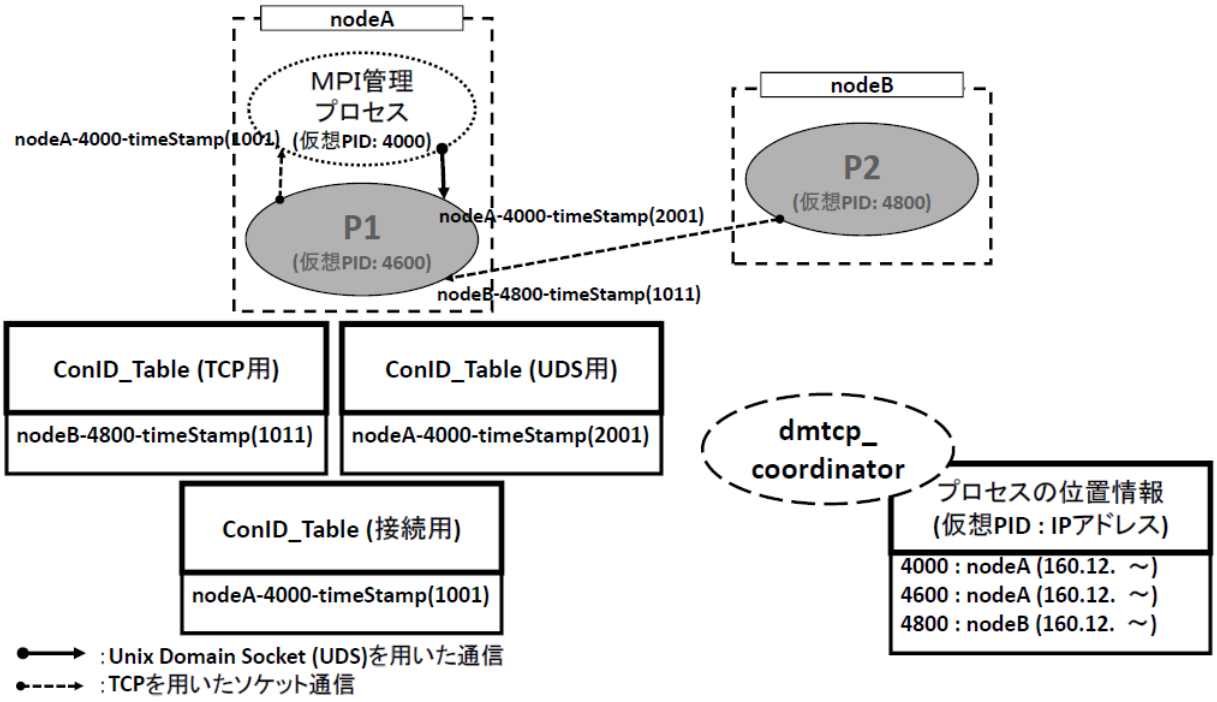

図 4.2 ConID_Table の具体例 (図中の P1 限定)

握できる．接続用の通信の ConID は 1 つのテーブルにまとめて保存される．しかし各プロセスがどのノードで起動しているかは各プロセスは把握していない．そこで ‘各ノードがどのノード (IP アドレス) で起動しているか’ という情報 (以降プロセスの位置情報) を dmtcp_coordinator が用意する処理を新たに加える．このプロセスの位置情報は各ノードで起動しているプロセスの仮想 PID と起動しているノードの IP アドレスが 1 対 1 対応している．図 4.2 に，ソケットドメインごとに分けて保存する ConID と dmtcp_coordinator が用意するプロセスの位置情報の具体例を示す．

図 4.2 では，1 つの MPI 管理プロセスと 2 つの MPI 並列実行プロセスが登場し，特に P1 が持つ ConID のテーブル情報 (ConID_Table) を例示する．また図 4.2 では受信側の ConID のみを表示する．図 4.2 の P1 は TCP を用いたソケット通信で接続要求を受ける通信が 1 つあるため，TCP 用 ConID_Table には 1 つの ConID が存在する．一方で UNIX ドメインソケットを用いた通信は MPI 管理プロセスから接続要求を受ける通信 1 つであるため，UNIX ドメインソケット通信用 ConID_Table には 1 つだけ ConID が存在する．また P1 は TCP を用いて MPI 管理プロセスへ接続要求を発行するため，接続用 ConID_Table には 1 つの ConID が存在する．また図 4.2 に示したような dmtcp_coordinator が持つプロセスの位置情報を STEP2. の開始前に各プロセスに送信する．各プロセスがプロセスの位置情報を取得して通信方法の変更し，STEP2. を開始するまでのフローチャートを図 4.3 に示す．

図 4.3 中の初めの条件分岐において，子プロセスのみ通信方法の変更を行う理由は並列分散処理を実際に行う子プロセス (MPI 並列実行プロセス) 間の通信性能が MPI アプリケーションの実行時間に大きな影響を与えるからである．次に各 ConID_Table 内の ConID を取り出し，dmtcp_coordinator から受信したプロセスの位置情報との比較を行い，通信方法の変更を行う．この変更は受信側と接続側両方で行い，その後 STEP2. を開始する．以上の実装方法で全ての要件を満たし，プロセス間通信の効率化機能を実現した．

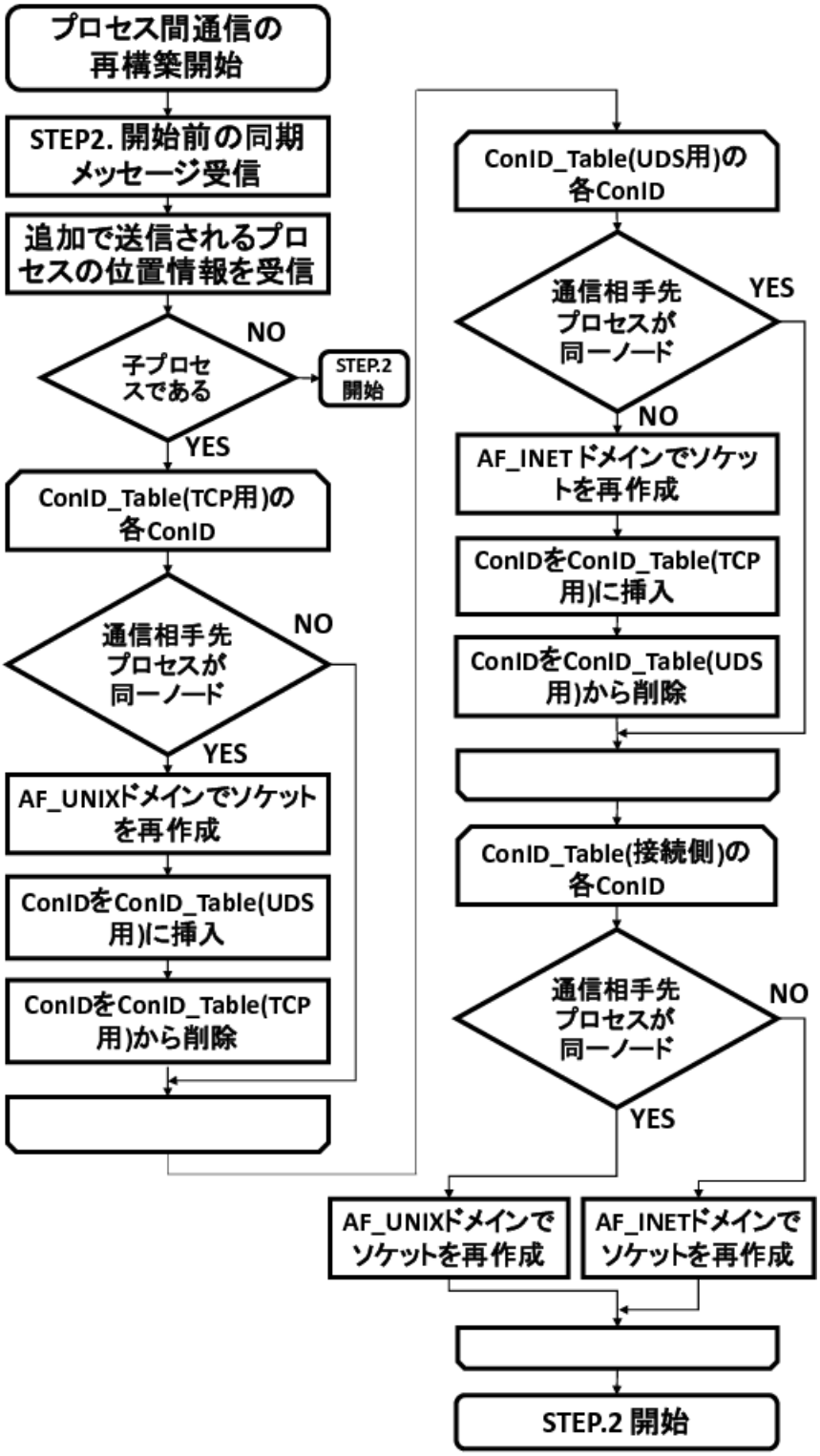

図 4.3　プロセスの位置情報取得から STEP2. を開始するまでのフローチャート

第5章　　性能評価

本章では実現した並列タスクの負荷分散機能とプロセス間通信の効率化機能の 2 つを評価する．また，効率的な並列分散処理の維持を実現する新たな手法のヒントがないかを探るため,「チェックポイント処理のオーバーヘッド」と「アプリケーション特性とプロセスの配置の関係」についても評価した．各評価で使用する評価用テストアプリケーションと評価結果の考察に使用するプロファイラについて述べたのちに評価結果を示す．また本評価において使用した DMTCP のバージョンは 2.3.1，Open MPI のバージョンは 1.6.4 である

5.1　テストアプリケーション

評価では N クイーンプログラムと NAS Parallel Benchmarks(NPB) プログラムの MPI 並列版を使用する．まず N クイーン (N-queen 問題) とは N × N のボード上に N 個のクイーンを互いに取りあわないように配置するパターンの総数を求める問題である．MPI 並列版プログラム [18] は各 MPI 並列実行プロセスは割り当てられた仕事が終了した時点で，マスタから仕事を割り当ててもらうマスタ・ワーカ方式を用いている．1 つのプロセスがマスタを担当し，残りのプロセスがワーカとして動作する．またワーカが行う仕事 (タスク) について述べる．MPI 並列版プログラム [18] では N × N のボードにおいて，上段から下方向に順番にクイーンを置いていく．この時最初の 4 個のクイーンの配置を確定し，残ったクイーンの組み合わせを試しながら解の数をカウントする処理を 1 タスクとする．クイーン数が 23 の場合を例とすると，調べるべきタスクは 64072 個存在する．ワーカはマスタから処理すべきタスク番号を受信し，計算から得た解の数をマスタに送信する．以上がこの MPI 並列版プログラム [18] のアルゴリズムの説明となる．本研究では並列タスクの負荷分散機能の効果を明らかにするために，この MPI 並列版プログラム [18] に変更を加えたプログラムを評価で使用する．この MPI 並列版プログラム [18] では，ワーカプロセスの実行時間が常に均等になるアルゴリズムである．例えばリスタート処理時にプロセスの移動によりリモートノードで起動するプロセスがそのノードのコア数を越えて性能が低下した場合，リモートノードのワーカプロセスが担当するタスクは減少し，性能が低下していないノードのワーカプロセスが多くのタスクを担当する．これにより，ワーカプロセス間のタスク処理量にばらつきが生じるが各プロセスのトータルの実行時間はほぼ均一となる．このように MPI 並列版プログラム [18] 中に既に負荷分散機能が存在する．そ

のため本研究において実現した並列タスクの負荷分散機能の効果を明らかにするために各ワーカプロセスに割り当てるタスクの数を固定にする．例えばクイーン数が 23，ワーカプロセスが 4 つの場合，総タスク数は 64072 であるため各ワーカプロセスは 1/4 である 16018 タスクを処理する．このタスク数はアプリケーションが終了するまで不変であり，動的に変化しない．本研究では以上のアルゴリズムに変更した MPI 並列版プログラム [18] を評価に使用する．

次に NPB プログラムについて述べる．NASA Ames Research Center で開発された並列コンピュータのためのベンチマークである．NPB は 5 つの Parallel Kernel Benchmarks と 3 つの Parallel CFD(Computational Fluid Dynamics) Application Benchmarks から構成される．各プログラムには問題のサイズのことなる 7 つのクラスが定義されている．7 つとは昇順でそれぞれ S(ample)，W(orkstation)，A，B，C，D，E である．その相違点は，基本的には問題サイズ (配列サイズや反復回数) の違いにある．評価では評価環境や評価条件 (プロセス数) に応じた問題サイズ (クラス) を設定とする．

5.2 MPI アプリケーションのプロファイル

本研究において実装した機能の効果を示すためにリスタート後の MPI アプリケーションのプロファイル結果があると評価の考察が深く行える．しかし DMTCP のリスタート後のゲストアプリケーション (本研究では MPI アプリケーション) のプロファイル結果を取得できるツールが存在しない．そこで一般的な MPI アプリケーションのプロファイラを用いて，考察の参考とする．プロファイルでは一度 DMTCP を使わずに MPI アプリケーションを実行し，プロファイル結果を得る．そのプロファイル結果からリスタート後の MPI アプリケーションの実行を分析する．フリーで使用できる MPI のプロファイルには SIM-MPI や IPM，mpiP が挙げられる．SIM-MPI は mpich にしか対応していない上に現在開発も進行していない．IPM も同様に現在開発も進行していない．

そこで現在も開発が進行している MPI アプリケーション用の軽量プロファイリングライブラリである mpiP(ver 3.4.1) を使用する [19] [20]．このライブラリは MPI 関数に関する統計情報のみを収集するため，少ないオーバーヘッドでプロファイルデータを作成する．プロファイルは MPI プロファイリングレイヤーを介して MPI 情報を収集し，プロセスごとにプロファイル結果を得ることができる．得ることができるプロファイル結果は主に以下の 4 つである．

プロファイル結果 1. MPI 並列実行プロセスにおける総実行時間に加えて MPI 関数の実行時間

プロファイル結果 2. アプリケーション内で最も時間を消費した上位 20 個の MPI 関数

プロファイル結果 3. 合計送信メッセージサイズが多い上位 20 個の MPI 関数

プロファイル結果 4. プロファイル結果 1. 3. をまとめ，プロセスごとに表示

プロファイル結果 1. ではプロセスごとの総実行時間と実行時間のうちに占める MPI 関数を実行した時間を示す．この総実行時間は MPI_Init() の終了から MPI_Finalize() の開始までの時間を示す．プロファイル結果 2. では mpiP が MPI アプリケーションのソースコード内の MPI 関数 1 つ 1 つに割り当てた識別子 (ID) をもとに，MPI 関数の実行時間の中で最も消費した上位 20 個の MPI 関数を示す．プロファイル結果 3. では合計送信メッセージサイズが多い上位 20 個の MPI 関数に加えて，それぞれの送信回数，MPI 関数の実行時間に占める割合も示す．プロファイル結果 4. ではプロファイル結果 1. からプロファイル結果 3. をまとめ，プロセスごとに示し，同じ MPI 関数の実行でも実行時間が長いプロセスなども特定できる．

5.3 2 つの新機能の評価における共通条件

評価では本研究において実現した 2 つの機能について，それぞれ適用した場合と適用しない場合における MPI アプリケーションがリスタートしてから終了するまでの実行時間を比較する．計測は

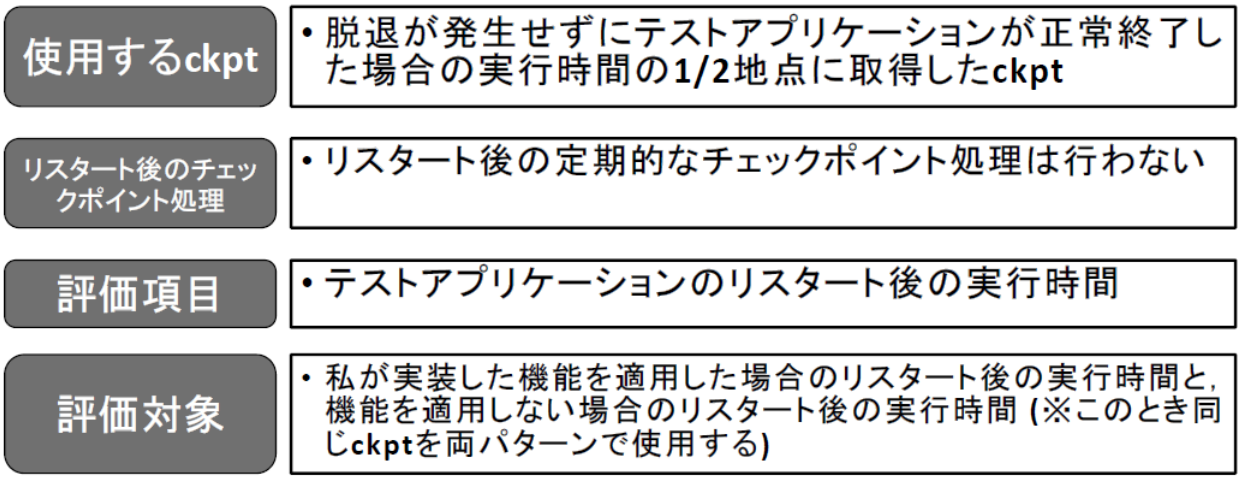

図 5.1 本研究の各評価における共通条件

dmtcp_coordinator が行う．計測範囲置は dmtcp_coordinator がユーザースレッドの再開用メッセージを各プロセスへ送信し，ゲストアプリケーションの計算終了後に管理下の全プロセスを終了させるまでである．以降この計測範囲の時間をリスタート後の実行時間として扱う．各評価における共通条件を図 5.2 に示す．

評価では実装した機能の効果を明らかにしたいため，リスタート後のチェックポイント処理は行わない．また現在 Android OS 上で DMTCP を動作させる環境を実現できていないため，Linux PC(CPU:Core i7 4770, 動作周波数:3.4GHz, コア数:4, メモリ:32GB) を利用する．評価ではこの Linux PC を 1 台から 3 台使用する．

性能が同じマシンを使用することで，PC の性能差ではなく本研究において実装した機能による影響のみを抽出する．

5.4 並列タスクの負荷分散の性能評価

本節では並列タスクの負荷分散機能の評価を行う．従来のノード単位での割り当てによってクラスタ全体の性能が著しく低下するのは，プロセスの再配置によってノードが備えるコア数を越え，1 プロセスが 1 コアを占有して活用することができないためではないかと予想する．よって評価ではノード単位での再配置によってあえて再配置先ノードが持つコア数を越えるかつプロセス単位での再配置によってノードが持つコア数を越えないようなシチュエーションを設定し，有線環境/無線環境それぞれで評価を行う．また，プロセス単位での並列タスクの負荷分散の効果の妥当性も明らかにする．

5.4.1 評価環境と評価方法 (有線接続)

比較対象は，リスタート処理時に DMTCP が従来から持つノード単位での再配置を行った場合とプロセス単位での再配置を行った場合を比較する．ノード間のネットワークスループットは約 940Mbps である．測定には iperf[21] と呼ばれるネットワークスループット測定用のフリーソフトウェアを用いた．本節の評価では 3 ノードで構成するクラスタシステムから 1 ノードが脱退し，リスタートするシチュエーションを想定する．3 ノードのうち 1 ノードは 3 プロセス起動し，残り 2 ノードは 2 プロセス起動する．脱退するノードは 3 プロセス起動しているノードであり，この 3 プロセスをノード単位で再配置すると評価に使用する Linux PC のコア数を越える．

次にテストアプリケーションには 5.1 節で紹介した N-queen 問題を解くプログラムを使用し，クイーン数は 19 である．

本研究において設定した評価環境で N-queen 問題プログラムを実行すると実行時間は 10 回平均で約 327.94sec となるため，N-queen 問題プログラムが起動してから 163sec 地点でチェックポイント処理を行い，この時生成した ckpt をリスタート処理に使用する．本節で設定した評価条件/評価方法をまとめて図 5.2 に，評価結果を図 5.3 に示す．

図 5.3 中の凡例は脱退ノードの並列タスク (3 プロセス) を残りの 2 ノードにそれぞれ何プロセス再配置したかを示す．一番左の棒グラフは従来のノード単位での再配置を示し，remote nodeA に全プロセス再配置した場合である．プロセス単位での再配置を行った場合が残りの 2 本の棒グラフがプロセス単位で任意のプロセス再配置した場合である．host node に 1 プロセス，remote nodeA に 2 プロセス再配置した場合の実行時間が一番短く，179.63sec である．これはこの再配置では host node, remote nodeA 両方のノードのコア数を越えておらず，各プロセスが各コアをそれぞれ占有して使用可能だったからである．次に host node に 2 プロセス，remote nodeA に 1 プロセス再配置した場合の実行時間が host node に 1 プロセス，remote nodeA に 2 プロセス再配置した場合の実行時間より増加しているのは host node のコア数を越えたためであると考える．また実現した並列タスクの再配置機

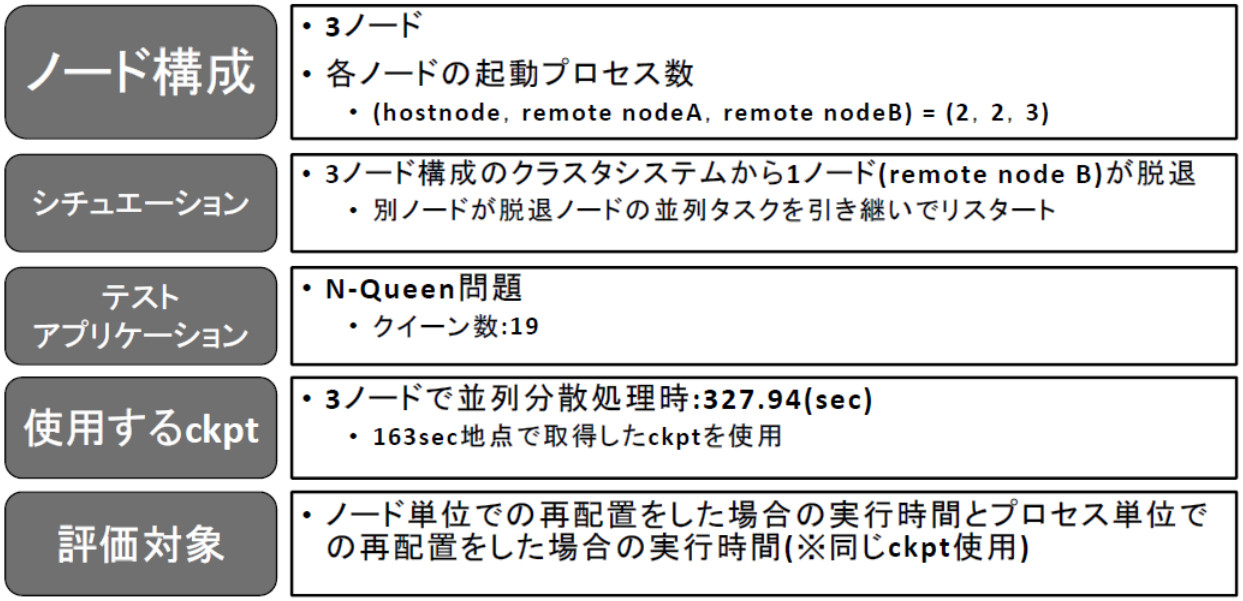

図 5.2 並列タスクの負荷分散機能の評価条件/評価方法

能では脱退ノードの orted を復元しないことも原因である．host node の orteurn が復元しない orted からの応答を得られず，1 コアを占有する．よって orterun と元々実行していた 2 プロセス，再配置した 2 プロセスの計 5 プロセスが起動し，host node のコア数を越える．次に remote nodeA にノード単位で再配置した場合は remote nodeA のコア数を越え，かつタスク番号の取得や計算結果の送信などの通信時間も加わるため，実行時間が一番長い 212.85sec となっている．図 5.3 より，remote nodeA にノード単位で並列タスクを再配置した場合と比較して，ノードのコア数を越えないようにプロセス単位で再配置を行うことで実行時間を 15.6%削減できた．

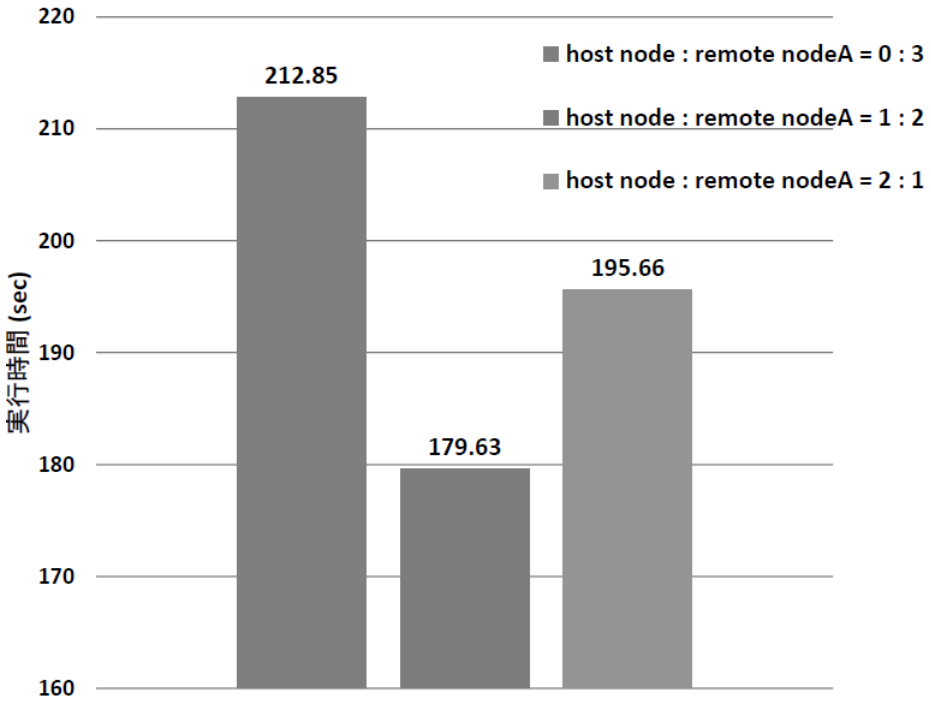

図 5.3　任意のプロセス数の再配置によるリスタート後の実行時間 (有線環境)

5.4.2　評価環境と評価方法 (無線接続)

5.4.1 節で行った評価を無線環境 (Wi-Fi:802.11b/g/n) で行い，無線環境下でもプロセス単位での再配置によって実行時間を削減できるか評価する．またプロセス単位での再配置による効果が妥当か判断する評価も行う．

ノード間のネットワークスループットは約 15Mbps である．テストアプリケーションには通信時間も含めて各プロセスの実行時間がほぼ同じである NPB プログラムの中の EP を使用する．一度 DMTCP を用いずにクラスタ上で EP を実行した場合のプロファイル結果 1. を図 5.4 に示す．

図 5.4 の Task は MPI 並列実行プロセスのランクを示し，MPITime は MPI 関数の総実行時間，MPI%はプロセスの全体の実行間に対して MPITime が占める割合を示している．Task 0 から Task 1 が host node の 2 プロセス，Task 2 から Task 3 が remote nodeA の 2 プロセス，Task 4 から Task 6 が remote nodeB の 3 プロセスである．図 5.4 の AppTime からも分かるように，各プロセスの実行時間はほぼ同じ (差異が 0.0020%) である．そして 5.4.1 節の評価同様，3 プロセス実行している remote nodeB が脱退し，2 プロセス実行している残りの 2 ノードに任意のプロセス数再配置した場合の実行時間を計測する．計測結果を図 5.5 に示す．

図 5.5 より，プロセス単位での再配置によって Host(host node) に 1 プロセス，Remote(remote nodeA) に 2 プロセス再配置した場合の実行時間が一番短く，ノード単位で再配置した場合と比較して少な

```
**EP(CLASS:D)**
@--- MPI Time (seconds) ----------
Task    AppTime(sec)    MPITime(sec)    MPI%
  0         496             0.844       0.17
  1         495             0.0802      0.02
 ------------------------------------------------------------
  2         495             64.7        13.07
  3         495             63.6        12.85
 ------------------------------------------------------------
  4         495             52.8        10.67
  5         495             52.1        10.53
  6         495             51.4        10.40
```

図 5.4　3 ノード構成のクラスタ上で実行した場合のプロファイル (EP)

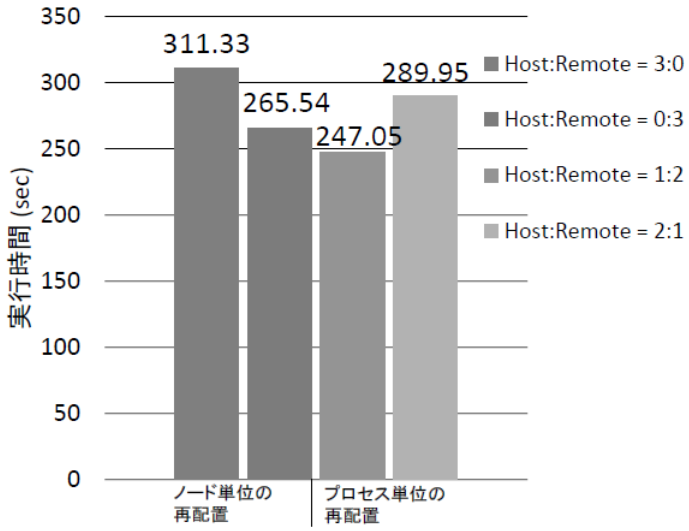

図 5.5　任意のプロセス数の再配置によるリスタート後の実行時間 (無線環境)

くとも 6.7%実行時間を削減できた．そのため，無線環境下においてもプロセス単位での再配置による並列タスクの負荷分散は実行時間の削減するために効果があることが確認できる [22].

次にプロセス単位での再配置を行った図 5.5 の評価結果が妥当かどうかの評価を行う．この評価ではクラスタに残った 2 ノードがもともと実行していたプロセス数に再配置したプロセス数を加えたプロセス数を Open MPI の machine file を用いて設定し，起動 & 実行した場合の実行時間を示す．図 5.5 において，Host にノード単位で再配置した場合を具体例として説明する．この時 Host がもともと実行していた 2 プロセスも含めて 5 プロセス実行し，Remote では 2 プロセス実行している．そのため，2 ノードで構成されるクラスタ上で MPI アプリケーション (EP) を起動する際に，machine file によって 1 ノードは 5 プロセス，もう 1 ノードは 2 プロセス実行するように設定し，再配置後のプロセス数を模擬する．そして DMTCP を併用せずに並列分散処理し，この時の実行時間を測定す

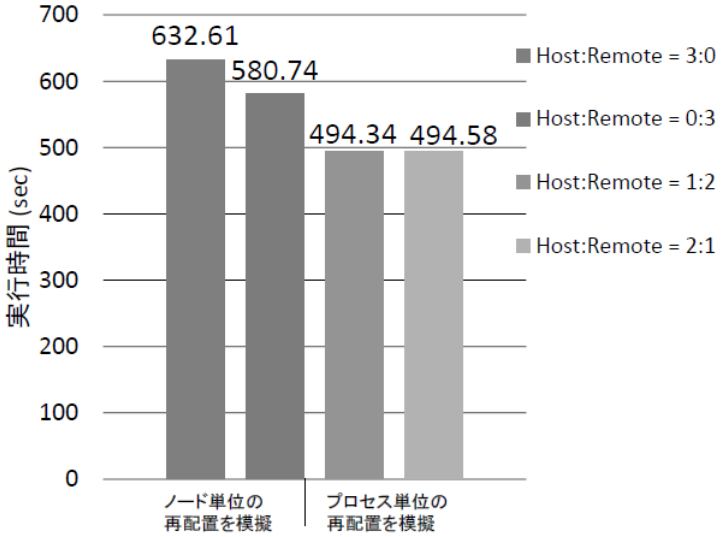

図 5.6　3 ノード構成のクラスタ上で実行した場合のプロファイル (EP)

る．つまりノード構成の動的変更が発生しないまま，アプリケーションを処理する．図 5.5 の凡例 (再配置したプロセス数) をもとに，この測定を行う．測定結果を図 5.6 に示す．

図 5.6 より，プロセス単位での再配置を行った場合の実行時間が一番短く，図 5.5 の評価結果と同じである．ノード単位での再配置を行った場合の実行時間がプロセス単位で再配置した場合と比べて長いのは図 5.5 の評価結果と同様，ノードのコア数を越えたためである．同じノード単位での再配置においても，Host に再配置した場合の実行時間の方が長いという結果も図 5.5 の評価結果と同じである．唯一図 5.5 の評価結果と異なるのは，Host に 2 プロセス，Remote に 1 プロセス再配置した場合の結果である．図 5.6 の評価結果では Host に 1 プロセス，Remote に 2 プロセス再配置した場合の実行時間の差はわずか 0.14sec であるのに対して，図 5.6 の評価結果ではその差は 42.9sec であった．原因は 5.4.1 節でも述べたが，実現した並列タスクの再配置機能では host node の orteurn が復元しない orted からの応答を得られずにスリープ状態とならないからである．図 5.6 中のプロセス単位での再配置を模擬した評価ではどの場合でも各ノードのコア数を越えないが，図 5.5 中の Host に 1 プロセス，Remote に 2 プロセス再配置した場合では orterun がスリープ状態にならないため，ノードのコア数を越えてしまう．orterun がスリープ状態に移行するように改善することで，図 5.5 と図 5.6 におけるプロセス数ごとの実行時間における相互の関係は完全に同じになると考える．orterun がスリープ状態に移行しない問題を含めたとしても，プロセス単位での再配置による効果は妥当な結果であると判断できる．

5.5 プロセス間通信の効率化の性能評価

本節ではプロセス間通信の効率化機能の評価を行う．本研究では特に単一ノード内の通信において，プロセス間通信の効率化機能を適用すると MPI アプリケーションの実行時間を削減できるのではないかと考える．単一ノードであるため，他ノードとの間の通信が発生せず，プロセス間通信の効率化機能の効果を測ることができる．また通信の多い MPI アプリケーションでは大幅な実行時間の削減が期待できる．一方で単一ノードではなく，複数ノードでのリスタート処理時にプロセス間通信の効率化機能を適用した場合についても測定する．複数ノードの場合はノード間の通信が発生し，通信時間の占める割合が増加する．この場合でもプロセス間通信の効率化機能により実行時間を削減可能か評価する．よって本節の評価ではテストアプリケーションを用いて，単一ノードでリスタートする場合と複数ノードでリスタートする場合の 2 つのシチュエーションを設定する．

5.5.1 評価環境と評価方法 (単一ノードの場合)

比較対象は，リスタート処理時にプロセス間通信の効率化機能を適用しノード内の通信を UNIX ドメインソケットを用いた通信に変更した場合と，適用せずに TCP を用いたソケット通信のままリスタートする場合である．評価では Linux PC を 2 台使用する．本節の評価では 2 ノードで構成するクラスタシステムから 1 ノードが脱退し，単一ノードでリスタートするシチュエーションを想定する．ノード間のネットワークスループットは約 940Mbps である．各ノードでは 2 プロセス起動し，リスタート後はホストノードが脱退ノードの並列タスクを引き継いで全 4 プロセス実行する．次にテストアプリケーションには NPB プログラムの 8 つのプログラムを使用し，全てクラス C を設定する．リスタートに使用する ckpt は，5.4.1 節と同様に，テストアプリケーション実行開始時のクラスタ構成において脱退が発生せず正常終了した場合の実行時間の 1/2 地点で取得した ckpt を使用する．本節で設定した評価条件/評価方法をまとめて図 5.7 に示す．またリスタートに使用する ckpt の取得地点について表 5.1 に示す．表 5.1 中の ‘チェックポイント処理地点’ はテストアプリケーションが起動してからの経過時間である．なおこの評価を評価 A と呼称する．

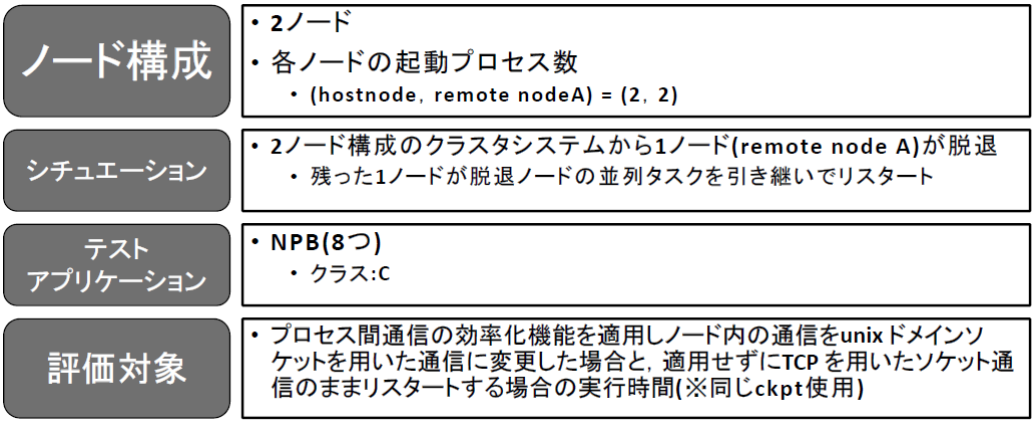

図 5.7　プロセス間通信の効率化機能の評価条件/評価方法 (単一ノード実行)

表 5.1　評価 A における各テストアプリケーションの ckpt 取得時間

NPB	脱退なしの実行時間 (sec)	チェックポイント処理地点 (sec)
CG	78	39
MG	20	10
SP	309	154
LU	171	85
BT	248	124
FT	228	114
EP	54	27
IS	22	11

評価結果を表 5.2，図 5.8 に示す．表 5.2 ではプロセス間通信の効率化機能を適用した場合としない場合のリスタート後の実行時間を示す．図 5.8 は表 5.2 をもとに，プロセス間通信の効率化機能を適用することで，効率化機能を適用しない場合に比べてどれくらい実行時間を削減できたかを示す．

表 5.2 評価 A におけるリスタート後の実行時間

NPB	効率化なし:リスタート後の実行時間 (sec)	効率化あり:リスタート後の実行時間 (sec)
CG	76.37	49.56
MG	38.06	21.98
SP	224.26	223.4
LU	201.6	190.0
BT	193.53	183.43
FT	58.21	55.84
EP	32.95	32.49
IS	4.21	3.57

図 5.8 より，CG と MG が実行時間の削減率がそれぞれ 35.1%，42.2%と大きく，SP と EP はほとんど削減されない．LU と BT は約 5%の削減率に止まっている．削減率が高い CG と MG は通信回数が多いと予想する．

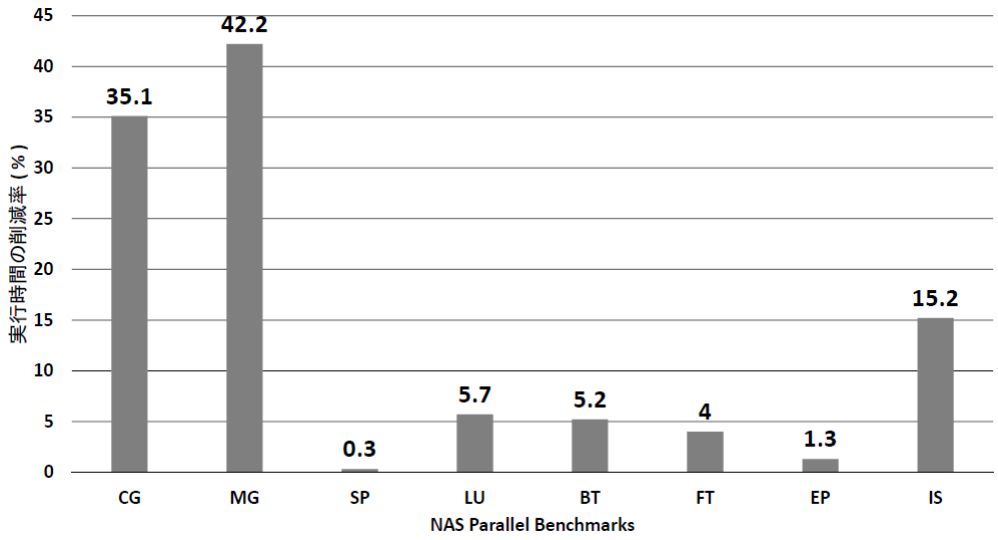

図 5.8　表 5.2 から導いた通信の効率化適用による実行時間の削減率

脱退前の状況である 2 ノードでクラスタを構成し，各 2 プロセス起動している中で mpiP を用いてプロファイルを取得し，主に MPI 関数の呼ばれる頻度やデータサイズ，通信形態について示す．はじめに一番削減率が大きい MG のプロファイル結果を表 5.3，5.4 に示す．

表 5.3 mpiP による MG のプロファイル結果 2.

MPI 関数	App%	MPI%
give3_() の 6 番目の MPI_Send	13.03	38.51
give3_() の 5 番目の MPI_Send	12.15	35.93
take3_() の 1 番目の MPI_Wait	4.58	13.54
give3_() の 4 番目の MPI_Send	3.38	9.99

表 5.4 mpiP による MG のプロファイル結果 3.

MPI 関数	Count(回)	Total(bytes)	Avrg(bytes)
give3_() の 6 番目の MPI_Send	2200	3.76e+08	1.71e+05
give3_() の 5 番目の MPI_Send	2200	3.76e+08	1.71e+05
give3_() の 3 番目の MPI_Send	2200	3.71e+08	1.69e+05
give3_() の 4 番目の MPI_Send	2200	3.71e+08	1.69e+05
give3_() の 2 番目の MPI_Send	2200	1.85e+08	8.39e+04
give3_() の 1 番目の MPI_Send	2200	1.85e+08	8.39e+04

表 5.3 中の App%はアプリケーション全体の実行時間に占める割合 (%) を示し，MPI%はアプリケーションが実行する MPI 関数の総実行時間に占める割合 (%) を示す．表 5.4 中の Count は一度のアプリケーション実行で実行するトータル回数である．Total は送受信するデータの総量 (bytes) を示す．Avrg は一度の送受信の平均データ量 (bytes) を示す．表 5.3，5.4 から MG は 1 対 1 通信の MPI_send 関数が主であり，通信回数も多い．

表 5.5　mpiP による CG のプロファイル結果 2.

MPI 関数	App%	MPI%
conj_grad() の 4 番目の MPI_Wait	14.31	45.29
conj_grad() の 3 番目の MPI_Send	10.39	32.9
conj_grad() の 3 番目の MPI_Wait	3.99	12.3
conj_grad() の 2 番目の MPI_Send	1.55	4.9

表 5.6　mpiP による CG のプロファイル結果 3.

MPI 関数	Count(回)	Total(bytes)	Avrg(bytes)
conj_grad() の 2 番目の MPI_Send	7600	4.56e+09	6e+05
conj_grad() の 3 番目の MPI_Send	7600	4.56e+09	6e+05
conj_grad() の 6 番目の MPI_Send	304	1.82e+08	6e+05

次に CG のプロファイル結果を表 5.5，5.6 に示す.

表 5.5，5.6 から CG は 1 対 1 通信の MPI_send 関数が主であり，通信回数も多い．実行時間の 1/2 地点でチェックポイント処理を行ったことを考慮すると，リスタート後は約半分の (3800 * 2 + 152) 回もの通信が行われ，(9302000000 / 2) bytes のデータ通信が発生したと表 5.4 からおおよそ予想できる.

表 5.7 mpiP による FT のプロファイル結果 2.

MPI 関数	App%	MPI%
transpose2_global() の Alltoall	73.70	99.91

表 5.8 mpiP による FT のプロファイル結果 3.

MPI 関数	Count(回)	Total(bytes)	Avrg(bytes)
transpose2_global() の Alltoall	88	1.18e+10	1.34e+08

一方で削減率が小さい FT や EP，SP のプロファイル結果を示す．FT のプロファイル結果を表 5.7, 5.8 に示す．

表 5.7，5.8 から FT はノード間通信の発生で通信付加が大きくなるプログラムであるが集団通信関数である Alltotall 関数が MPI 関数の実行時間のほぼ 100%を占める．集団通信の場合は全てのプロセスが送受信可能になるまで通信が行えないため，一部のプロセス間通信が高速になったとしても全体の実行時間への影響は大きくない．加えて集団関数である Alltotall 関数の呼び出し回数も MG や CG に比べて極めて少ない．

表 5.9　mpiP による EP のプロファイル結果 2.

MPI 関数	App%	MPI%
embar() の 1 番目の Allreduce	0.92	97.55

表 5.10　mpiP による EP のプロファイル結果 3.

MPI 関数	Count(回)	Total(bytes)	Avrg(bytes)
embar() の 1 番目の Allreduce	4	320	80

EP のプロファイル結果を表 5.9，5.10 に示す.

表 5.9，5.10 から EP ではほとんど通信が発生しない．MPI 関数の 97%を占める Allreduce 関数も合計 4 回しか呼ばれない．そして通信データ量も MG や CG に比べて極めて少ない.

表 5.11 mpiP による SP のプロファイル結果 2.

MPI 関数	App%	MPI%
y_solve() の 1 番目の Waitall	11.81	49.21
copy_faces() の 1 番目の Waitall	4.72	19.67
z_solve() の 1 番目の Waitall	4.01	16.70

表 5.12 mpiP による SP のプロファイル結果 3.

MPI 関数	Count(回)	Total(bytes)	Avrg(bytes)
y_solve() の 2 番目の Isend	1604	1.81e+09	1.13e+06
z_solve() の 2 番目の Isend	1604	1.81e+09	1.13e+06
x_solve() の 2 番目の Isend	1604	1.81e+09	1.13e+06
copy_faces() の 2 番目の Isend	1608	8.44e+08	5.25e+05

SP のプロファイル結果を表 5.11，5.12 に示す.

表 5.11，5.12 から SP では通信回数，通信データサイズも大きいがプログラムに占める通信時間の割合が計算時間に比べて少ない．一番通信時間がかかる MPI 関数でもアプリケーション全体に対して 11.81%の割合しかない．また通信データサイズが大きいほど TCP を用いたソケット通信と UNIX ドメインソケットを用いた通信の転送レートの差が小さくなることも原因として考えられ，通信データサイズが大きくかつノンブロッキング通信を用いた通信であるため，通信の効率化による効果が見られなかった.

最後に LU，BT，IS についてのプロファイルを示す.

表 5.13　mpiP による LU のプロファイル結果 2.

MPI 関数	App%	MPI%
exchange_3_() の 1 番目の Wait	3.99	27.11
exchange_1_() の 2 番目の Recv	3.19	21.66
exchange_1_()1 番目の Recv	1.71	11.62

表 5.14　mpiP による LU のプロファイル結果 3.

MPI 関数	Count(回)	Total(bytes)	Avrg(bytes)
exchange_1_() の 1 番目の Send	508	5.33e+08	1.05e+06
exchange_1_() の 2 番目の Send	508	5.33e+08	1.05e+06
exchange_1_() の 3 番目の Send	508	5.33e+08	1.05e+06
exchange_1_() の 4 番目の Send	508	5.33e+08	1.05e+06

LU のプロファイルを表 5.13，5.14 に示す．

表 5.13，5.14 から LU は CG や MG ほどデータ送受信回数は多くはない．そして 1 対 1 通信を行う通信形態であることも分かる．同じ 1 対 1 通信の通信形態で CG や MG よりも削減率が低いのはデータ送受信の回数が多くないことが原因であると考える．

表 5.15 mpiP による BT のプロファイル結果 2.

MPI 関数	App%	MPI%
y_solve.f:1 番目の wait	6.62	28.55
copy_faces.f:1 番目の Waitall	6.04	26.04
z_solve.f:1 番目の wait	4.11	17.71

表 5.16 mpiP による BT のプロファイル結果 3.

MPI 関数	Count(回)	Total(bytes)	Avrg(bytes)
y_send_solve_info_() の 1 番目の Isend	804	1.3e+09	1.61e+06
x_send_solve_info_() の 1 番目の Isend	804	1.3e+09	1.61e+06
z_send_solve_info_() の 1 番目の Isend	804	1.3e+09	1.61e+06

BT のプロファイル結果を表 5.15，5.16 に示す．

表 5.15，5.16 から BT はノンブロッキング通信を用いた 1 対 1 通信の通信形態であると分かる．CG や MG ほどデータ送受信回数は多くなく，LU より多い通信回数である．またアプリケーションに対する MPI 関数の実行時間の割合も少ないため，通信負荷より計算負荷が大きいプログラムであると言える．そして LU より通信回数が多いが削減率が LU の 5.7%とほぼ変わらない 5.2%である理由は通信方法がノンブロッキングであることが考えられる．

表 5.17　mpiP による IS のプロファイル結果 2.

MPI 関数	App%	MPI%
rank() の 1 番目の Alltoallv	73.37	94.49
rank() の 1 番目の Allreduce	3.88	5.00

表 5.18　mpiP による IS のプロファイル結果 3.

MPI 関数	Count(回)	Total(bytes)	Avrg(bytes)
rank() の 1 番目の Allreduce	44	1.81e+05	4.12e+03
rank() の 1 番目の Alltotall	44	176	4

IS のプロファイル結果を表 5.17，5.18 に示す.

表 5.17，5.18 から IS の通信形態は集団通信である．FT と同様に集団通信を行うため，FT とほぼ等しい削減率が予想されるが図 5.8 より，15.2%の実行時間の削減である．しかし表 5.2 より実行時間が約 4sec と短いため，通信の効率化機能の影響とは言いきれない.

5.5.2 評価環境と評価方法 (複数ノードの場合)

評価項目と比較対象は 5.5.1 節と同じである．5.5.1 節と異なる点はノード構成だけである．本節の評価では 3 ノードで構成するクラスタシステムから 1 ノードが脱退し，複数ノード (2 ノード) でリスタートするシチュエーションを想定する．3 ノードのうち 1 ノードは 2 プロセス起動し，残り 2 ノードは 1 プロセス起動する．ノード間のネットワークスループットは約 940Mbps である．脱退するノードは 2 プロセス起動しているノードであり，この 2 プロセスをノード単位で再配置した場合でも評価に使用する Linux PC のコア数を越えないため，実行時間の差はプロセス間通信の効率化機能による影響のみであると考える．本節で設定した評価条件/評価方法をまとめて図 5.9 に示す．なおこの評価を評価 B と呼称する．評価結果を表 5.20，図 5.10 に示す.

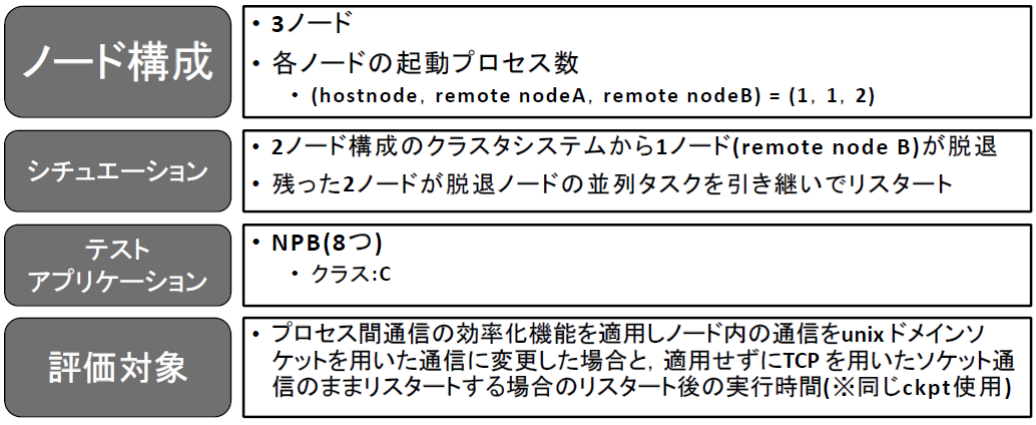

図 5.9 プロセス間通信の効率化機能の評価条件/評価方法 (複数ノード実行)

表 5.19　評価 B における各テストアプリケーションの ckpt 取得時間

NPB	脱退なしの実行時間 (sec)	チェックポイント処理地点 (sec)
CG	92	46
MG	21	10
SP	256	128
LU	169	84
BT	222	111
FT	193	96
EP	54	27
IS	19	9

表 5.20　評価 B におけるリスタート後の実行時間

NPB	効率化なし:リスタート後の実行時間 (sec)	効率化あり:リスタート後の実行時間 (sec)
CG	62.16	61.93
MG	26.52	15.78
SP	158.82	154.33
LU	102.31	92.55
BT	128.73	126.47
FT	118.89	118.79
EP	27.35	27.26
IS	18.59	18.03

表 5.20 ではプロセス間通信の効率化機能を適用した場合としない場合のリスタート後の実行時間を示す．図 5.10 は表 5.20 をもとに，プロセス間通信の効率化機能を適用することで，効率化機能を適用しない場合に比べてどれくらい実行時間を削減できたかを示す．

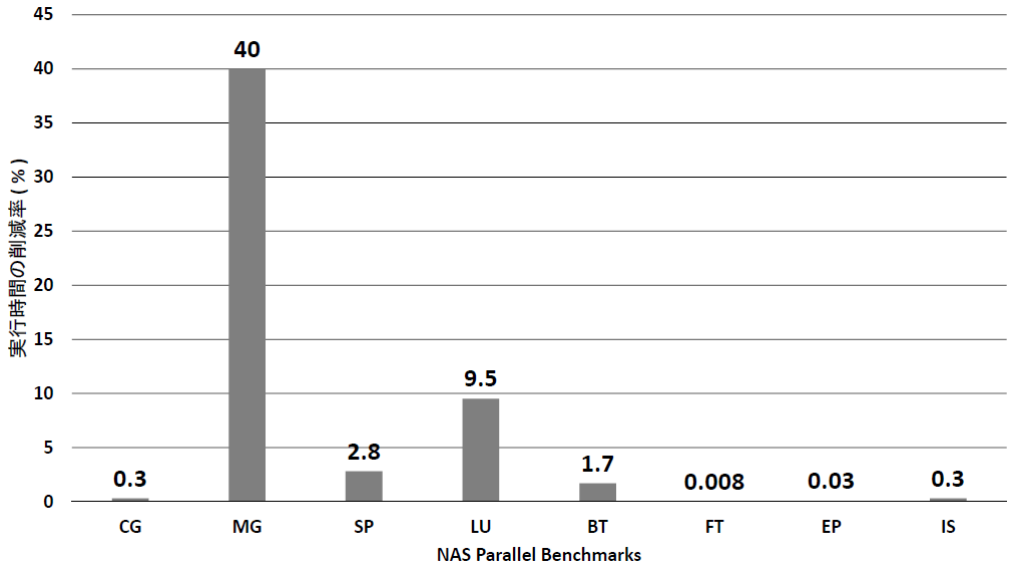

図 5.10　表 5.20 から導いた通信の効率化適用による実行時間の削減率

図 5.10 から MG だけ実行時間が大きく削減できた．全体として，ノード間の通信が発生したため，その通信がボトルネックとなり，ノード内通信を高速化しても効果が少なかったと考える．FT は評価 A の場合と同様，集団通信が主となるため，ノード内通信を高速化しても全体の実行時間に大きな影響は与えない．IS は評価 A では実行時間を 15.2%削減できたが，評価 B では同じ集団通信を行う FT 同様に削減率が 1%を下回る．一方で評価 A では MG と同様に大幅に実行時間を削減できた CG の削減率は 0.3%であった．一つは MG に比べて通信回数が多いため，ノード間通信部分の通信時間の影響が大きくなったと考える．また LU が CG よりも削減率が低い原因は，CG に比べて通信回数が少ないためノード間通信部分の通信時間の影響が小さいと考える．

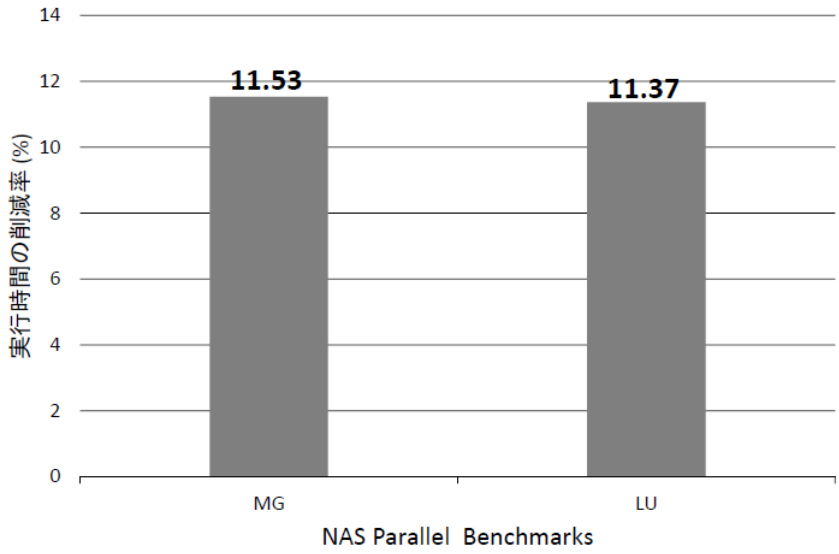

図 5.11　無線通信環境下:通信の効率化適用による実行時間の削減率

次に無線通信環境下での評価を行う．評価 B ではノード間の通信は有線としていた．そこで，評価 B の通信環境を Wi-Fi(802.11b/g/n) とした場合の評価 (評価 C) を行う．ノード間のネットワークスループットは約 15Mbps である．評価 C では通信の効率化による効果が見れた MG，LU を対象に評価した．評価 B 同様に，削減率を図 5.11 に示す．

図 5.11 より，無線通信環境下においても，通信の効率化によって実行時間を削減できる [22]．MG は評価 B においては削減率 40.0%を示していたが評価 C では 11.53%となり，削減率が大きく低下している．原因は通信帯域が 15Mbps と乏しくなり，評価 B に比べてノード間の通信負荷がよりボトルネックになったからである．加えて MG は LU と比較して，1 回の通信データサイズは同程度だた通信回数が約 55 倍 (評価 A のプロファイル結果より算出) となり，通信回数が著しく多い．

5.6 評価 1:チェックポイントオーバーヘッド

本システムではクラスタ内のどのノードがいつ脱退しても並列分散処理を維持するために，定期的にチェックポイントデータを取得している．そこでテストアプリケーションの実行時におけるチェックポイントデータの取得が与える影響を明らかにする [17] [22]．そこでチェックポイントデータの取得を行わない場合におけるテストアプリケーションの実行時間と，任意の間隔でチェックポイントデータの取得を行った場合のテストアプリケーションの実行時間を計測する．そして，チェックポイントデータ取得によるオーバーヘッド，つまりチェックポイントデータの取得を行わない場合と比較してどの程度実行時間が増加したかを示す．テストアプリケーションには N-queen 問題プログラムと NPB プログラムの 1 つである IS を使用する．この 2 つの MPI アプリケーションを選んだ理由はアプリケーションにおける通信負荷 (合計の転送サイズ) の差によってオーバーヘッドが異なると予想したからである．IS は N-queen 問題プログラムと比較して，通信負荷が大きいアプリケーションとなっている．N-queen 問題プログラムのクイーン数は 19，IS のクラスは C である．クラスタの構成要素は 2 ノードであり，各ノード 2 プロセス実行の 4 並列である．ノード間は Wi-Fi で接続され，ノード間のネットワークスループットは約 15Mbps である．評価結果を図 5.12 に示す．

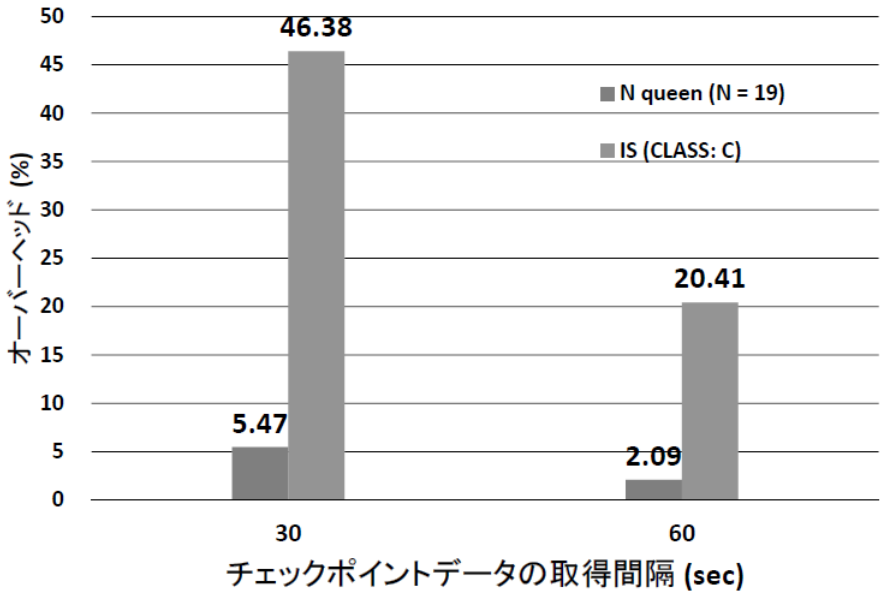

図 5.12　各チェックポイントの間隔におけるオーバーヘッド

図 5.12 より，IS はチェックポイントの間隔が 30 秒の時，チェックポイントデータの取得を行わない場合と比較して実行時間が 46.38%増加している．また，チェックポイントの間隔が同じ場合，合計のデータ転送サイズが大きいアプリケーションである IS の方がオーバーヘッドが大きい．チェックポイントデータのサイズを比較すると，N-queen 問題プログラムにおけるチェックポイントデータのサイズは 4.7Mbyte，一方 IS におけるチェックポイントデータのサイズは 267Mbyte であった．以上の結果からチェックポイントの間隔が短いほどオーバーヘッドが大きい．特に保存するプロセスイメージのサイズが大きいアプリケーションはデータ転送サイズが大きい，つまり通信負荷が大きいアプリケーションにおいて，オーバーヘッドが大きいことが分かる．そのため，チェックポイントの間隔を長くよることでオーバーヘッドを抑えることができる．ただし，チェックポイントデータを長くしすぎた場合，最悪のケースとして最初のチェックポイントが完了する前にノードの脱退等によるアプリケーションの中断が発生したとき，並列分散処理の継続が不可能となる．

そのため，最適なチェックポイントの間隔を考える必要もある．図 5.12 より，通信負荷の小さい N-queen 問題プログラムではチェックポイントの取得間隔を倍にすることでオーバーヘッドを約 2 分の 1 に削減できる．そのため，チェックポイントの取得間隔を 60sec の倍である 120sec にすることで，オーバーヘッドを 2 分の 1 の約 1%に抑えることができ，アプリケーションの実行にほぼ影響を与えることはない．また通信負荷の大きい IS においても取得間隔を倍にすることでオーバーヘッドを約 2 分の 1 に削減できることが図 5.12 より考えられる．チェックポイント取得間隔が 60sec のときにオーバーヘッドは 20.41%であるため，取得間隔を 2 の 3 乗 (8 倍) にすることでオーバーヘッドも 8 分の 1 の約 3%に抑えることができると考える．

5.7 評価 2:アプリケーションの特性とプロセスの配置

5.4.1 節の評価では，プロセス単位での再配置によって，ノードのコア数を越えないようにプロセスを再配置し，実行時間を削減できることを示した．しかしノードのコア数を越えた場合の方が，コア数を越えた場合の実行時間より短い場合があると考える．例えば，データの通信回数・通信量が多いアプリケーションである．無線等の通信帯域が乏しい環境ではノード間の通信時間が長くなるため，ノードのコア数を越えることを許容して最小限のノードで実行した場合の方がアプリケーションの実行時間が短くなる可能正があると考える．そこでアプリケーションの並列度を変えずに，単一ノード上でノードのコア数を越えて実行した場合と，2 ノードで構成されるクラスタ上で各ノードのコア数を越えないようプロセスを配置した場合を比較する．前者ではノード間通信が発生せず，後者ではノード間通信が発生する．そして 1 ノードで実行した場合と比較して，2 ノードで実行した場合の実行時間が何倍増加したか，その増加率 (式 5.1) を図 5.13 示す．

(2 ノードの場合の実行時間−1 ノードの場合の実行時間)/1 ノードの場合の実行時間 = 増加率 (倍) (5.1)

評価で使用するノードのコア数は 4 であることを考慮して，並列度を 8 とし，8 つの MPI 並列実行プロセスを起動する．このとき単一ノード上で実行した場合は各コアは 2 プロセス実行し，2 ノードで構成されるクラスタ内の各ノードでは 1 プロセスが 1 コアを占有する．データの通信回数・通信量がそれぞれ異なる N-queen 問題プログラムと NPB プログラムを評価に使用する．N-queen 問題プログラムのクイーン数は 19 とする．ただし NPB プログラムにおいて，並列度を 8 プロセスに設定できる 6 つのプログラム (MG, CG, EP, LU, FT, IS) を使用する．また，mpiP を用いたプロファイルから平均データ転送サイズが大きい上位 20 個の MPI 関数と，各 MPI 関数における転送データサイズが得られるため，各プログラムの合計転送データサイズも調査する．

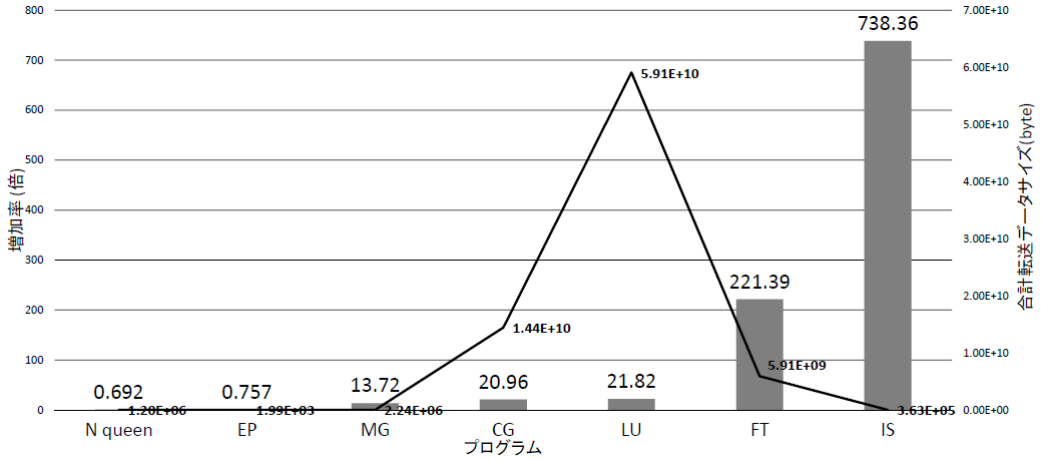

図 5.13　2 ノード実行における 1 ノード実行時からの増加率

図 5.13 の折れ線グラフは各プログラムにおける合計転送データサイズ (単位:byte) を示している．図 5.13 より，データの通信回数・通信量がほとんどない EP のように，データの合計転送サイズが小さいプログラムでは，2 ノードで並列分散処理した場合の方が実行時間が短い．一方，データの合計転送サイズが大きいプログラムは 2 ノードで並列分散処理した場合，単一ノードで並列分散処理した場合と比較して少なくとも実行時間が 13 倍以上となっている．特に通信形態が集団通信である IS と FT の増加率は著しく，IS は実行時間が 221.39 倍となっている．IS の合計転送データサイズは FT よりも小さいが IS の方が増加率が高い．EP も同じ集団通信かつ合計転送データサイズが小さいが，MPI の集団通信関数をコールする回数が 1 回と IS に比べて小さい．そのため，EP は 2 ノードで並列分散処理した場合，単一ノードで並列分散処理した場合と比較して速度向上が見られた．よって，次のことが明らかとなった．

1.1 対 1 通信を行うアプリケーションでは合計転送データサイズが大きいほどノード間通信部分の通信時間が長くなり，ボトルネックとなる

2. 通信形態が集団通信であり，集団通信の回数が多いアプリケーションは単一ノードで並列分散処理する

この評価より，アプリケーションにおける通信時間・計算時間を考慮して，プロセスを配置・クラスタの構成を考えることは有効であると考える．

第6章　おわりに

本研究では Android 端末で構成する Android クラスタにおいて，ノード構成の動的変更が発生後，ノード間の並列処理の負荷バランスが不均衡であり，またプロセス間の通信が非効率なまま並列処理を継続する場合があった．そのため，クラスタの性能を最大限に維持するためのの機能を実現した．そして実現した機能によって，ノード構成の動的変更後のアプリケーションの実行時間を削減可能か評価を行った．

本論文ではまず先行研究である Android クラスタシステムの構成や機能を要約した．そして Android クラスタにおいて，ノード構成の動的変更時にノード間の並列タスクの負荷に不均衡が生じること，加えてプロセス間の通信方法が非効率なままで並列分散処理を再開する問題があることを述べた．そこで並列タスクの負荷分散と通信の効率化によってこれらの問題を解決できると考えた．

検討した 2 つの機能は並列分散処理継続メカニズムを実現する DMTCP と呼ばれるチェックポインティングソフトウェアを変更することで実現できる．そこで一般的な DMTCP の処理 (特にリスタート処理) のアルゴリズムや仕様を述べつつ，検討した機能の実装上の課題を踏まえ，機能を実現した．

1 つ目の並列タスクの負荷分散は並列タスクの再配置をノード単位ではなくプロセス単位での再配置を可能とする機能である．この機能によって，脱退したノード内の並列タスクを複数のノードに分散して再配置でき，ノード間における並列タスクの負荷の不均衡を抑え，クラスタ全体の性能低下を緩和することができる．

2 つ目は並列タスクの負荷分散に伴い，非効率な通信が発生する場合があるため，プロセスの再配置時にプロセス間の通信方法を切り替える機能である．この機能では通信を行う各プロセスが起動しているノードに応じて，通信方法を TCP を用いたソケット通信から，より高速な UNIX ドメインソケットを用いた通信に切り替える．例えば TCP を用いたソケット通信を行っていたプロセス同士が同一ノードで起動した場合は UNIX ドメインソケットを用いた通信に切り替え，通信の効率化を実現する．

そして 2 つの機能によって，ノード構成の動的変更後も最大限の性能を維持可能かどうかを評価した．特にノードの脱退が発生した場合に，実現した機能を適用することにより性能低下をどの程度緩和できるかを有線接続環境また無線接続環境それぞれの場合において評価した．並列タスクを

プロセス単位で再配置する機能の評価では有線環境/無線環境ともに実行時間を削減できることを確認した．特に N-queen 問題プログラムを対象に評価したところ，従来のノード単位での再配置時に比較して実行時間を最大 15.6%削減できることを確認した．

通信の効率化の評価では NPB プログラムの 8 つを対象に，単一ノードでリスタートした状況ではプロセス間通信を効率化しない場合と比較して，実行時間を最大 42.2%削減できた．またノード間通信が発生する状況でも同様の評価を行った場合，単一ノードでの評価時に効果が確認できた NPB プログラムにおいても通信の効率化の効果が減少し，実行時間を削減できるプログラムが少なくなった．原因はノード間の通信がボトルネックとなり，ノード内の通信を高速化したとしても実行時間削減への影響が小さいためであると考える．また無線通信環境下において，同様にノード間通信が発生する状況で評価を行い，最大で 11.53%実行時間を削減できることを確認した．

さらに，ノード構成の動的変更後も最大限の性能を維持するクラスタシステムを実現するために，本システムの評価を 2 つ行った．まず 1 つ目は並列分散処理継続メカニズムである DMTCP のチェックポイントデータ取得によるオーバーヘッドを調査した．チェックポイントの間隔を 30 秒に設定し，NPB プログラムの IS を評価したところ，実行時間が 46.38%増加した．チェックポイントの間隔が短いほどオーバーヘッドは大きいため，取得間隔を長くすることでオーバーヘッドを抑制できると考える．また通信の効率化機能の評価から，アプリケーションにおける通信負荷によって機能使用における効果が異なることが明らかになっている．そこで 2 つ目に，プロセスの配置方法にもアプリケーションにおける通信負荷が影響を与えるか評価を行った．通信負荷が異なる複数のアプリケーションの実行時間を計測したところ，転送データサイズが小さいアプリケーションでは複数ノードにプロセスを配置することで高速化が可能である．一方転送データサイズが大きいアプリケーションはノード間通信がボトルネックとなる．単一ノードで実行した場合と比較して実行時間が最大 738.36 倍となるアプリケーションも存在した．そこでアプリケーションの通信負荷，つまり転送データサイズによっては，ノードが持つコア数を越えることを許容して単一ノードに全プロセスを配置した方が高速に並列分散処理が行えることを確認できた．

本研究の意義についてまとめる．先行研究では，ノード構成の動的変更時にノード間の並列処理の負荷バランスに不均衡が生じ，プロセス間の通信方法が非効率なまま並列分散処理が行われる場合があった．しかし本研究によって，ノード構成の動的変更後もクラスタシステムの性能を最大限維持するための機能を新たにシステムに実現し，先行研究の課題を改善した．

今後の課題はチェックポイントデータ取得によるオーバーヘッドの最小化とノード間通信による通信負荷の軽減である．チェックポイントデータ取得によるオーバーヘッドを削減するためには取得間隔を長くする必要があると本論文では述べた．しかしあらかじめアプリケーションの実行時間をシステムは把握していないため，1 度アプリケーションを実行し，実行時間を記録し，その実行時間をもとに取得間隔を設定することでオーバーヘッドを極力削減し，かつチェックポイントデー

タの取得を確実に行うことが可能であると考える．次に通信負荷の軽減についてだが，本研究における評価から，ノード間通信のコストが大きいことが確認できている．より高速に並列分散処理を維持するためには，ノード間通信による通信負荷が軽減するようなプロセスの配置を行う機構等が必要ではないかと考える．例えば本論文の評価から転送データサイズが大きいアプリケーションは 1 ノードで実行するようにプロセスを配置する方が高速な並列分散処理が行えることが確認できている．

謝辞

本研究の機会を与えていただき，また，日頃から貴重な御意見，御指導いただいた，大津 金光准教授，大川 猛助教，横田 隆史教授，馬場 敬信教授に深く感謝致します．そして，本研究において多大な御力添えを頂いた，鋼グループをはじめとする研究室の方々に感謝致します．

参考文献

[1] (株) 矢野経済研究所, “携帯電話の世界市場に関する調査を実施 (2016 年)”, https://www.yano.co.jp/press/pdf/1607.pdf, 2016 年 11 月 4 日, (2017 年 2 月 7 日参照.)

[2] Nikola Rajovicxz, Paul M. Carpenterx, Isaac Geladox, Nikola Puzovicx, Alex Ramirezxz, and Mateo Valerox, “ Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC?”, Proc. of SC’13 International Conference on High Performance Computing, Networking, Storage and Analysis, pp.1-12, 2013.

[3] Mir Muhammad Juno, Ali Raza Bhangwar, and Asif Ali Laghari, “Grids of Android Mobile Devices”, ICI-CTT, pp.1-3, 2013.

[4] Ketan B. Parmar, Nalinbhai N. Jani, Pranav S. Shrivastav, and Mitesh H. Patel, “jUniGrid: A Simplistic Framework for Integration of Mobile Devices in Heterogeneous Grid Computing”, International Journal of Multidisciplinary Sciences and Engineering, Vol. 4, No. 1, pp.10-15, 2013.

[5] 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.

[6] Felix Bsching, Sebastian Schild, and Lars Wolf, “DroidCluster: Towards Smartphone Cluster ComputingThe Streets are Paved with Potential Computer Clusters”, In Distributed Computing Systems Workshops (ICD-CSW), pp.114-117, 2012.

[7] 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”, 電子情報通信学会技術研究報告, Vol.114, No.155, pp.143-148, 2014.

[8] Gartner, Inc. , “Gartner Says Chinese Smartphone Vendors Were Only Vendors in the Global Top Five to Increase Sales in the Third Quarter of 2016”, http://www.gartner.com/newsroom/id/3516317 (2017 年 2 月 7 日参照.)

[9] 合田憲人, “3 章 分散コンピューティングシステム”, 電子情報通信学会 「知識ベース」, http://www.ieice-hbkb.org/files/06/06gun_05hen_03.pdf, (2017 年 2 月 7 日参照.)

［10］ 日本アイ・ビー・エム システムズ・エンジニアリング株式会社, “グリッド・コンピューティングとは何か Globus Toolkit ではじめるグリッドの基礎”, ソフトバンク パブリッシング株式会社, 2004.

［11］ Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall, “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”, Proceedings, 1th European PVM/MPI Users’ Group Meeting, pp.97-104, 2014.

［12］ Jason Ansel, Kapil Arya, and Gene Cooperman, “DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop”, 23rd IEEE International Parallel and Distributed Processing Symposium, IPDPS2009, pp.1-12, 2014.

［13］ Xi Qian, Guangyu Zhu, and Xiao-Feng Li, “Comparison and Analysis of the Three Programming Models in Google Android”, First Asia-Pacific Programming Languages and Compilers Workshop (APPLC), pp.1-9, 2012.

［14］ 片桐 孝洋, “スパコンプログラミング入門 並列処理と MPI の学習”, 東京大学出版会, 2013.

［15］ Joshua Hursey, Timothy I. Mattox, and Andrew Lumsdaine, “Iterconnect agnostic checkpoint/restart in Open MPI”, HPDC ’09: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, pp.1-10, 2009.

［16］ Paul H Hargrove and Jason C Duell, “Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters”, Journal of Physics: Conference Series, Vol.46, pp.494-499, 2006.

［17］ Yuki Sawada, Yusuke Arai, Kanemitsu Ootsu, Takashi Yokota, and Takeshi Ohkawa, “An Android Cluster System Capable of Dynamic Node Reconfiguration”, Proc. Seventh International Conference on Ubiquitous and Future Networks (ICUFN 2015), pp.689-694, 2015.

［18］ 吉瀬 謙二, 片桐 孝洋, 本多 弘樹, 弓場 敏嗣, “qn24b:N-queens の解を計算するベンチマークプログラム”, FIT2004 第 3 回情報科学技術フォーラム, 第 4 分冊, No.O-011, pp.389-392, 2004.

［19］ mpiP: Lightweight, Scalable MPI Profiling, http://mpip.sourceforge.net/ (2016 年 11 月参照.)

［20］ Jeffrey S. Vetter, and Michael O. McCracken, “Statistical Scalability Analysis of Communication Operations in Distributed Applications”, Proceeding PPoPP ’01 Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp.123-132, 2001.

［21］ iPerf - The ultimate speed test tool for TCP, UDP and SCTP, https://iperf.fr/, (2017 年 1 月参照.)

[22] Yuki Sawada, Yusuke Arai, Kanemitsu Ootsu, Takashi Yokota, and Takeshi Ohkawa, "Performance of Android Cluster System Allowing Dynamic Node Reconfiguration", Wireless Personal Communication, DOI 10.1007/s11277-017-3978-9, 2016.