Vol.2021-OS-153 No.5
2021/7/20

Many-core における Multi-SLAM に向けた ROS の Node の割当手法の検討

福井　誠人[1]　　石綿　陽一[2]　　大川　猛[3]　　菅谷　みどり[1]

概要：Society5.0 では，複数台の自律移動ロボットによるサービスの普及が期待される．自律走行ロボットでは SLAM(Simultaneous Localization and Mapping)技術による周辺地図の生成が必須であるが，処理負荷が高く，これを近距離で支援するために高性能なエッジサーバを配置することが期待されている． 一般的に SLAM の実装は，移動距離推定や地図作成などの複数のソフトウェアプロセスにより構成される．このことからエッジサーバに Many-core プロセッサを用いることで効率的に複数台のロボットの SLAM を支援できると考えられる．しかし，Many-core プロセッサをエッジサーバに活用した場合の有効性は明らかではない．そこで，本研究では，SLAM を主なアプリケーションとした場合 Many-core プロセッサの活用を検討する．
活用にあたり，汎用的な仕組みを目指し，SLAM のベースとなるミドルウェアである ROS(Robot Operating System)単位での割り当てを検討した．ROS はプロセスを Node として位置透過的に CPU に配置できる一方，非同期分散であることから，Node の実行時間の取得が困難である．本研究は，ROS の Node の実行時間の正確な計測に基づく Many-core への処理割り当てを目的とする．そのためまず，ROS 上で Node の実行時間を正確に計測するツールを設計実装した．次に，その結果に基づいた割り当てを実施した．その結果， ROS Node 正確な計測が実現でき，さらにキャッシュを考慮して ROS Node をプロセッサに適切に割り当てることで，1.6%の実行性能の向上を実現した．

キーワード：ROS, many-core, Edge computing, SLAM

1. はじめに

現在， 自律移動ロボットは,駅や空港といった公共機関の建物内で案内や清掃などの目的で導入されている[1]．これらの自律走行式の清掃ロボットは，あらかじめ設定されたルートに従って，複数台で清掃タスクを行う[2]．また，警備ロボットでは，3D LiDAR を搭載し，周辺環境の把握や落とし物，不審物を認識するサービスを提供する[3]．また，本サービスにおいては複数のロボットが，同時並行的に稼働し，不審者の発見,周辺の異常の検知，通報などを行う[3]．こうしたロボットの多くは，自律走行等の実現のために，大量の環境情報をセンシングし，そのデータを解析することでアクチュエーションを行う．

このように現在では多くの自律移動式のロボットが複数台で活用されている．しかし，こうしたロボットの大半は単純な制御方法を使用しているか人間が遠隔操作しているかのいずれかである[4]．また，センシングデータは計算機資源の制約などから制御計算をした後は保存されずに破棄されていた．しかし，近年ネットワークに接続されたドローンなどによりロボットが撮影したデータがクラウドに送信されるなどのサービス[5][6]が普及することで，センシングデータはクラウドサービスへ転送保存されるようになり，より効率的な自律移動などが実現できるようになってきた．一方，多数のロボットからのデータ送信が発生した場合，クラウドへ処理が集中し，クラウドサーバの過負荷や，遠距離のクラウドへの無線等のネットワークの遅延が課題となる[7]．

問題を解決する一つの方法として，エッジコンピューティングが検討できる[8]．エッジコンピューティングでは，デバイスに近いエッジ側に存在するセンサーノードに対し，近距離でのリアルタイム応答を目的とし，大量のセンサーノードに対し応答性を重視した設計が提案される[8]．クラウドとクライアントマシンであるロボットの間に，豊富な計算資源を持つエッジサーバを配置することで，クラウドサーバに集中していた負荷の分散や，ネットワーク通信による遅延や不安定性の解消をすることが可能となる [8]．

このように，エッジサーバは，ロボットなどのデバイスから送られる多量のデータを処理することから，サーバに求められる信頼性と，応答性を両立させる高い処理性能が求められる．特にロボットにおいては，大量の並列処理や低遅延での応答性能が求められることから，実用的なエッジの実現方法を明確にすることが期待される．しかし，まだ十分ではない．

本研究では，複数台ロボットの稼働支援およびエッジの性能と信頼性を必要とする実アプリケーションとして，SLAM (Simultaneous Localization and Mapping)を用いる．SLAM はロボットに搭載されている LiDAR 等によるリモートセンシングや車輪の回転速度などのセンサ値を使ってロボットの周囲の地図の作成，およびその空間上での位置を特定する技術である．SLAM の技術の多くはミドルウェアである ROS (Robot Operating System) 上で動作する．ROS は pub/sub による非同期分散システムとして実装されており[9], 複数の非同期処理を行う SLAM の構築を容易にすると同時に，複数台にわたる計算機処理を可能としている．例えば，ロボットのセンサから取得された大量のデータを

1 芝浦工業大学
2 東海大学
3 株式会社 Ales

ROS のメッセージとしてエッジサーバに送信し，エッジサーバは，そのデータを高性能な並列処理により同時並行的で処理する方法などが考えられる．これにより，複数のロボットが接続した場合であっても，応答性の向上が期待できる．この際に，エッジサーバ側での高性能な並列処理が重要となる．

エッジサーバにおいて，SLAM のような汎用的なアプリケーションを並行的に処理することで性能を向上させる方法として，Many-core 計算機の使用が考えられる．Many-core 計算機は，汎用的な CPU を用いて，並列性を生かしたアプリケーション動作において，キャッシュのコヒーレンシを活かすことで，処理に適した応答性，性能を達成できると考えられる．しかし，Many-core において，SLAM を実用的に利用する方法は十分検討されていない．並列性を生かした処理を行うためには，プロセスごとの処理の見積もりを正確にたて，コアにプロセスを配置することが望ましいが，非同期型の実装である ROS においては，プロセスがデータの送受信を行った正確な時間のトレースが困難である課題がある．

そこで，本研究では Many-core を用いた複数台ロボットのエッジサーバにおける応答性．処理性能向上を目指し，ROS のトレースを正確に行うことで Many-core プロセッサを用いたタスク管理を実現することを目的とした．予備実験として Many-core プロセッサに処理を割り当てた際の実行時間について基礎的な評価し，プログラム自体の並列性を考慮してプロセッサを割り当てた場合一定の時間で処理を行う事が可能であることを確認した．

また，複数台のロボットに向けたエッジサーバのタスク管理の目的の実現のために，ROS を使用した際にプロセスの実行を計測するための仕組みを実装し，実際に実行をプロセッサに割り当てたときの効果について評価を行った．10 台のロボットからのデータをエッジサーバに集約して SLAM を行うシミュレーションを行ったところ提案した手法では 1.6%の実行性能の向上を実現した．

本論文の構成を以下に示す．2 章では ROS についての背景知識について述べる．3 章で Many-core プロセッサに関する先行研究や簡易な評価について述べる． 4 章で本研究の提案をする．5 章で提案システムの評価及び SLAM を用いて ROS におけるプロセッサ割り当てについて評価する．最後に 6 章でまとめる

2. SLAM(Simultaneous Localization and Mapping)について

2.1 複数台のロボットにおける地図作成

ロボットの自律走行や，近年では自動運転に SLAM (Simultaneous Localization and Mapping) 技術が用いられている[10]．SLAM は，ロボットなどに搭載されているカメラや LiDAR センサからのデータをもとに空間上での自己位置の推定と，同時に周辺環境の地図の作成する技術の総称である．SLAM にはいくつかのアルゴリズムがあるが本研究ではその中で Gmapping を取り上げる．これは LiDAR センサからのデータを使う 2 次元 SLAM アルゴリズムで，動作モデルに従って粒子のサンプリングを行い観測に基づいて各粒子の重みを更新しその後地図を更新する．更新後は重みが最大となる粒子の状態値とその地図をその時刻での推定値とする．精度の高い計算結果を得るためには多くの粒子を必要とすることからアルゴリズムの計算複雑性は増加する．本研究では複数台のロボットが協調して SLAM 処理を行う．複数台のロボットによる SLAM ではまず，各ロボットがセンサデータを使いそれぞれのロボットの場所に応じた地図を分担して作成する．その後，それぞれが作成した地図を組み合わせることで一つの地図を作る．ここでは多数のロボットで非同期かつ並列的に SLAM 処理が行われることから，gmapping での計算複雑性は増大し，その結果，統合的な処理負荷は比例的に増加する．

2.2 ROS (Robot Operating System) とは

ROS(Robot Operating System) [7]はロボットアプリケーション開発のためのオープンソースミドルウェアライブラリである．ROS におけるプログラムの最小単位は Node で Node は，OS におけるプロセスである．

ROS での Node 同士は Topic と呼ばれる名前付き通信経路を使った Publisher/ Subscriber メッセージングモデルでデータ通信を行う．Node 間の関係は，図 1 の有向グラフであらわされる．図 1 の表記は rqt グラフ [12]と呼ばれ，ROS 上での Node 同士の相互関係を把握するために用いられる．

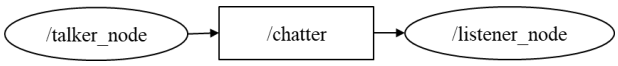

図 1 ROS での基本的な通信

この図では Node を楕円, Topic は四角形でそれぞれ表現する．Node にはデータを送信する Publisher とそれを Subscribe し，更新があったときに受信及びデータの処理を行う Subscriber を作成した．その他に接続を待ち処理の結果を返す Service があるが本論文では対象としていない．図 1 では 2 つの Node として，Publisher の/talker_node と Subscriber の/listener_node を作成した．また，これらの Node が Topic /chatter を介して通信する様子を示した．

ROS では，通信時に相手の Node について，その Node が位置する IP アドレスなどの物理情報は隠蔽されており，必要な Topic のみ分かっていれば通信を記述できる．このように，システム全体が分散システムとして利用できるように，設計されている．この仕組みを実現するために ROS1 では，物理情報のマッピングを，統一的に管理する ROS Master を構成し，各 Node は ROS Master に物理位置等の通

信に必要となる情報を問い合わせることで，通信相手を見つけることができる．

Subscriber は監視している Topic に新しいメッセージが送られたときに，メッセージを処理する Node にイベントが発行され，それを元に Topic データを読みに行くイベント駆動となっており，これにより非同期処理を実現している．開発者が，アプリケーションを実装する際には，Subscriber 側でメッセージを受け取る際に，実行する処理をコールバック関数として記述する．これにより， Topic にデータが送られ，それを Subscriber が処理する際に，該当するコールバック関数が呼び出される．

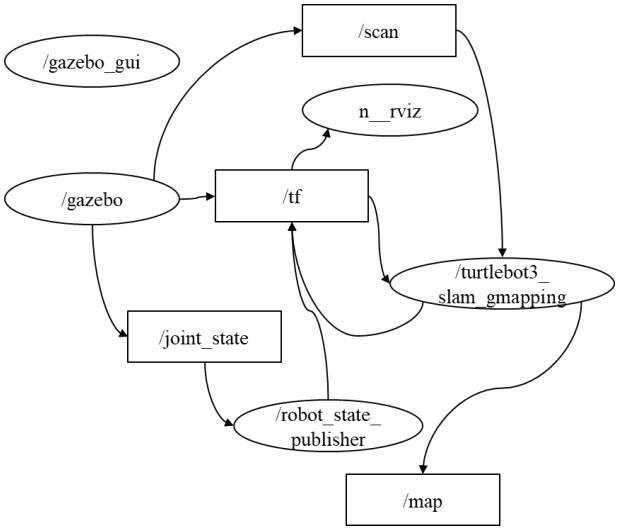

図 2 SLAM 実行時の rqt グラフ

図 2 に SLAM シミュレーションを行ったときの rqt グラフを示した．先ほどのサンプルとは異なり複数の Node, Topic が表示されている．これらの Node が複数のマシンで非同期に処理を行い，マップを作成する．図中の/gazebo Node は SLAM の物理演算シミュレータであり，シミュレーション環境内に位置するロボットのセンサ情報を Publish している．SLAM の地図生成のための計算処理は /turtlebot3_slam_gmapping Node が担当し，計算した結果を /map として出力する．

これらの Node は ROS のネットワークで接続されている複数台のマシンに分散して実行することができ Topic はネットワークを介して送受信される．

2.3 ROS アプリケーションの実行時間計測の課題

ROS では，詳細なアプリケーションの実行時間を計測することが難しい．SLAM を例として考えたとき，ロボットからセンサデータを送信した時刻と，SLAM がマップを出力した時刻は ROS のツールで取得することができる．しかし，ロボット上のセンサデータの送信 Node から SLAM を実行する Node までの通信時間が取得できない．

そのため処理と通信にそれぞれかかる時間を区別して計測することは困難である課題がある． 図 3 では ROS で取得可能な時刻と実際の Node の実行時間を表した．ROS で取得可能な時刻はメッセージ送信時刻であることから図中 T_0, T_2 のみである．しかし実際には T_0 から T_1 までの通信にかかる時間を考慮する必要がある．このときの Node の実行時間 ΔT はメッセージ到着時刻 T_1 から T_2 の時間であるがこれを直接計測する方法は ROS には現状実装されていない．

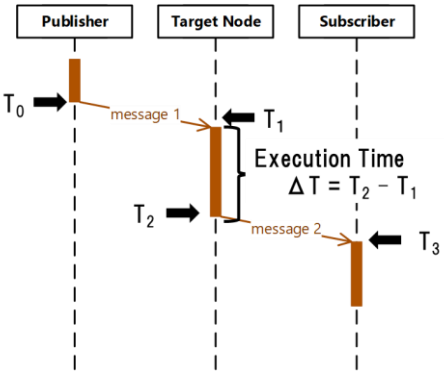

図 3 実際の Node の処理時間

2.4 rosbag

ROS アプリケーションのための代表的な計測ツールに rosbag [13]がある．これはメッセージが送信された時間を記録することができる一方，処理にかかった時間を計測する仕組みは提供されていない．例えば，同じ Topic を Subscribe する Node が複数あるとき，それぞれの Node がメッセージを受信し，処理を開始した時刻は Node ごとに異なる．これは ROS が非同期分散システムであることに起因する．そのためメッセージが送信された時刻のみでは ROS アプリケーションのコンポーネントごとの詳細な実行時間の正確な計測が困難であるという課題がある．

3. Many-core プロセッサのタスク管理の課題

3.1 先行研究

Many-core プロセッサをアプリケーションが使用する際の管理手法には，既に数多くの手法が提案されている．

谷本らは並列アプリケーションを Many-core で実行する時に競合とアプリケーションのスケーラビリティを考慮したスケジューラを提案した[14]. 谷本らは，複数のレベルのスケーラビリティを持つ並列アプリケーションを同時に実行する際に，アプリケーションの並列性を計測し，それをもとに全体でパフォーマンスが最も良くなるようメモリ階層の競合回避を考慮して適切な割り当てコア数を決定した．競合回避のためにこの研究では最下位キャッシュとメモリコントローラにおける競合に着目しプロセッサダイをアプリケーションへの割り当ての最小単位としている．この結果，主対象とする比較的スケーラビリティの低いアプリケーションの組みに対して Linux スケジューラに対して提案法の有効性を示した．しかし，具体的なアプリケーションへの適用は十分ではない．

三好らはプログラム内でのデータ転送量を考慮した Many-core プロセッサのタスクのコア割り当て手法を提案した[15]. メモリ管理を明示的に行う必要がある M-Core アーキテクチャにおいてノード間でデータを共有する場合 DMA 転送を用いてデータ通信を行う．このときにタスクをコアに割り当てた場合，適切な配置でないと実行サイクル数が増加する．このことからタスクのコアへの割り当てが実行時間に大きく影響するとし，C プログラムから生成された実行コードを事前に実行し，得られた各コア間のデータ転送量からデータ転送行列を作成する．また，あらかじめ用意した特定のアーキテクチャにおけるコア間通信のレイテンシ行列を使いコアへの割り当てを最適化した C プログラムコードを生成し， SimMc を用いた有効割り当てを発見し，性能向上を実現した．しかし，通信経路の衝突やコア-DMA 間のバンド幅による通信速度の律速により性能向上しないケースもある課題がある．

しかしながら，これらの Many-core のタスク管理手法について実際に即した複数台のロボットアプリケーションの並列実行などについてはまだ応用されていない．そこで，本研究では， Many-core を用いた複数台ロボットのエッジサーバにおける応答性．処理性能向上を目指し，ROS のトレースを正確に行う方法を検討する．また，予備実験として Many-core プロセッサに処理を割り当てた際の実行時間について基礎的な評価を行い，プログラム自体の並列性を考慮してプロセッサを割り当てる手法について検討する．

3.2 予備評価

Many-Core を実際のアプリケーションに適応させるためには， Many-Core におけるキャッシュを考慮した配置が必要となる．Many-core において汎用的なアプリケーションを用いて CPU を利用する場合の基礎的な CPU の利用についての調査はあまり公開されていない．そこで，本研究ではまず，この課題を遂行するにあたり，基礎的な性能評価を実施するものとした．

3.2.1 予備評価: Many-Core での CPU 利用の課題

エッジサーバでのシステム構成に先立ち，many-core プロセッサへの処理割り当てに関する基礎的な性能評価を行うことを目的として計測実験を行った．評価のため簡易な算術計算を行うベンチマークプログラムを作成した．これをプロセッサに割り当て，実行時間を計測した．プログラムは fork() システムコールを使い，複数のプロセス生成を行う．各プロセスはモンテカルロ法を用いて円周率を計算する．この時，プロセス間での同期や通信は発生しない．一度に生成するプロセス数，またその時に割り当てたプロセッサ数をそれぞれ変化させ，生成したプロセス数とプログラムに割り当てるプロセッサ数との関係を調査した．

表 1 予備評価 1: 実験環境

CPU	Intel Xeon Gold 6230 (20Core, 2.1GHz, 27.5MB cache) x 2
RAM	DDR4-2933 REG ECC 16G x 12
OS	Ubuntu 18.04.5

評価に使用した計算機を次の表 1 にまとめた．今回使用した CPU ではインテルハイパースレッディング・テクノロジーが利用可能であったが，タスクにプロセッサの割り当てを行ったときに物理コアとロジカルコアの差異が生まれるため正確に実験できない可能背を考慮してこの機能は使用しなかった．実験ではプログラムに割り当てたプロセッサ数を 1 から実験環境での最大数の 40 を割り当てて実行した．また一度に生成するプロセス数を 1 から 100 プロセスまで変化させ, それぞれの条件での実行時間を計測した．本実験ではプログラムの起動から終了までの実時間を対象として評価をまとめた．

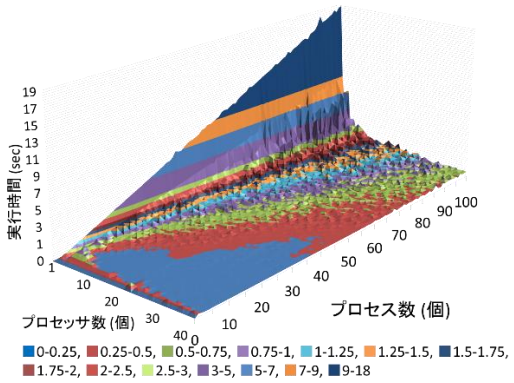

図 4 予備評価: 実験結果(3 次元グラフ)

実験の結果を 3 次元グラフにプロットした（図 4）．図 4 中では実行時間を z ラベル，プロセッサ数，プロセス数をそれぞれ x ラベル，y ラベルにグラフを示す．また，このうち 1, 10, 20, 40 プロセッサでの実行結果を図 5, 図 6, 図 7, 図 8 にそれぞれに示す．

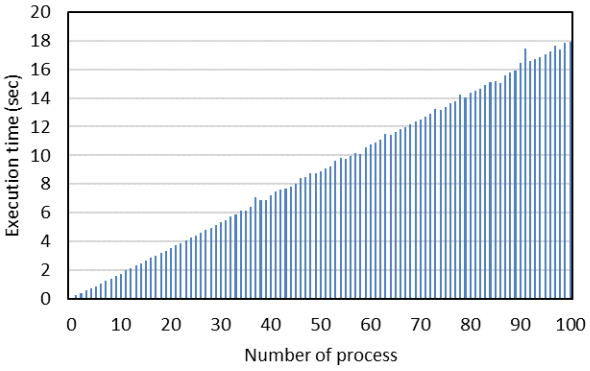

図 5 予備評価: 1 プロセッサで実行したときの結果

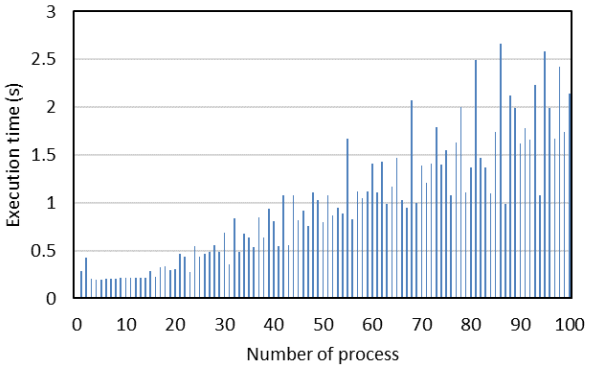

図 6 予備評価: 10 プロセッサで実行したときの結果

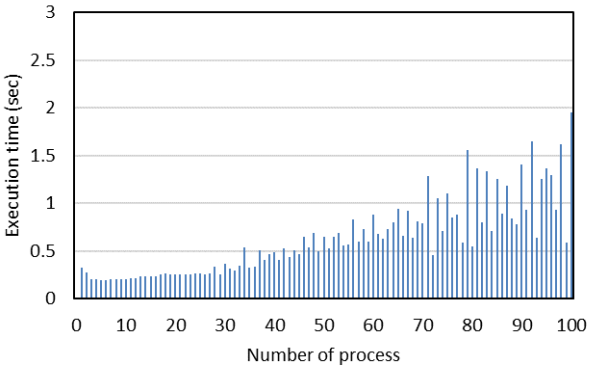

図 7 予備評価: 20 プロセッサで実行したときの結果

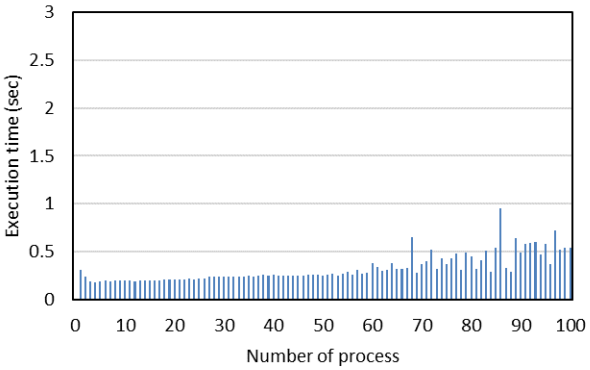

図 8 予備評価: 40 プロセッサで実行したときの結果

本評価実験ではプロセス間通信や，同期は行なっていない．このことから，プロセス数の増化は，純粋に処理負荷の増加といえる．これにより負荷を増加させた場合に，実行時間は線形的に増加すると考えられる．また同時に生成するプロセス数に着目すると，プロセス数が割り当てたプロセッサ数より少ないときに実行時間が横ばいである．このことからプログラムの並列性と同等，もしくはそれを上回るプロセッサを割り当てると，処理を一定時間で完了することができると考えられる．

4. 提案

4.1 課題

これまでの課題をまとめると(1) ROS でのプロセス (Node) の実行を正確に計測することが難しい. ROS が複数台のマシンで稼働することを想定した非同期分散システムであることやこのときの通信方式によりこれらの情報を収集する機能は十分提供されていない．このことから詳細な実行時のデータを，リアルタイムにプロセッサコアの割り当て制御に生かすことは現状困難である．また，(2) ROS においてプロセッサの割り当ては考慮されていない．特に Many-core プロセッサを持つマシンでのプログラムの実行を考えたときにそのキャッシュを考慮することは処理性能を向上させるために重要とされている一方で，ROS では Linux の標準スケジューラを使う事からキャッシュのコヒーレンシなどは考慮できていない．我々は，予備調査を実施し, その結果から Many-core でプログラムの並列性に応じたプロセッサ割り当てを考慮することで性能向上が見込めることがわかった．しかし，複数台ロボットのための処理性能の向上を目指す Many-core プロセッサの割り当て手法については検討されていない．

そこで本研究の目的は Many-core エッジサーバにおける複数台ロボットアプリケーションの処理性能向上の実現のためのタスク管理の指針を得ることとした．目的の実現にあたり，これまでの課題(1), (2) に対して次の 2 つの提案を行う．

4.2 提案 1 ROS Node の実行時間の正確な計測

Many-core プロセッサでの ROS の Node の実行を正確にトレースするために rosbag より詳細に Node の実行をトレースするツールを設計・実装を行う．特に本研究では複数のロボットアプリケーションが並列で動作したときの実行効率を向上することを目的としていることから，メッセージ受信後に Node で実行される処理について詳細な計測ができるツールが必要である．そこで，並列で動作する複数の Node に対して測定することができる処理時間計測ツールを作成した．

4.3 実装

本研究では評価の対象とした SLAM の実装言語の一つである, ROS の C++ 向けの実装 roscpp を拡張し実装した．

図 9 に roscpp を用いた開発で Topic を Subscribe する際に呼ばれる関数を含むソースコードの一部を示した．処理の順として最初にアプリケーションから (1) NodeHandle::subscribe() メソッドが呼ばれる．このメソッド内部では SubscribeOptions のインスタンスを作成し，渡された引数を SubscribeOptions のフィールドに設定する．次に(2)TopicManager::subscribe(), (3) addSubCallback() メソッドが順に呼ばれる．ここではすでに同名の Node が Subscriber として登録されていないかの確認やその時のコ

ンフリクトの解消を行う．(4) Subscription::addCallback() では先ほどまでの SubscribeOption から必要なフィールドを取り出し CallbackInfoPtr に設定する．(1) から(3) までの間でではコールバック関数の MD5 ハッシュ値を用いて識別していたが，ここで removal_id という別の ID を振り MD5 ハッシュは使われなくなる．最後に (5) CallbackQueue::addCallback() メソッドでコールバック関数を実行可能な状態にする．このようにして ROS 内部では Subscribe を実行する．

```
// 1) Subscribe to a topic.
NodeHnadle::subscribe("topic", 1000, function_ptr);
      SubscribeOptions ops;
      ops.topic = topic
      ops.callback = function_ptr;

// 2) Check if same Node exists.
TopicManager::subscscribe(ops);
// 3) Check if same Node exists.
TopicManager::addSubCallback(ops);

// 4) This function enclose `ops` to `info`.
Subsicription::addCallback(ops.topic, ops.md5sum, …);
kInfoPtr info(boost::make_shared<CallbackInfo>());
      info->helper_ = helper;
      info->… = …;
      removal_id = (uint64_t)info.get();
 /*
 *At this time, ops.md5sum is not inherited by info,.
 * So I mapped ops.md5sum and removed_id to identify
 * which topic it belongs to when callback is called.
 */

// 5) Set callback functon.
CallbackQueue::addCallback(info->subscription_queue_,
removal_id);
```

図 9 roscpp での Subscribe 時の実行関数群（一部抜粋）

本研究では，先ほどの方法で ROS では Topic の更新時にコールバック関数の呼び出しを可能とした．途中でコールバック関数を特定するために使われるキーが異なるため，そのままでは処理の呼び出し時にトレースできないことが分かった．そこで，計測ツールではコールバック関数の登録時に識別するために使われるハッシュ値と呼び出し時に使われる ID を対応付けて管理した．こうしたことであるタイミングでどの Subscriber, Topic に紐づいた関数が実行されたのかをトレースすることを可能とした．

これらを保存するためのデータ構造として下図 10 に示す NodeMeta 構造体を実装した．この構造体ではまず Subscribe した対象 Topic から ROS 内部で管理するために使われているハッシュ値，ID を保存した．また，提案 2 で Node のプロセッサ割り当てを行う事を可能にするために Node のプロセス ID や現在のプロセッサ割り当て状況を記録するための CPU 集合を記録する．本論文の実装ではこうした Node のメタデータを線形リストで扱うことで複数の Node を管理できるようにした．

また，コールバック関数の呼び出し前後で clock_gettime() 関数を使いその差分を計算することで Node での処理にどれほどの時間がかかったかを計測できるようにした．

```
struct NodeMeta
{
   std::string topic;
   std::string md5sum;
   uint64_t removal_id;
   pid_t pid;
   FixedQueue burst_times;
   std::int64_t burst_time;
   cpu_set_t *current_affinity;
   int max_cpuid;
   NodeMeta *next;
}
```

図 10 NodeMeta 構造体

4.4 提案 2: ROS Node の Many-core プロセッサ割り当て

ROS では，先行研究における Many-core プロセッサ向けのタスク管理手法は適用されていない．そこで，本研究では，複数台ロボットのために応答性と計算資源を提供するエッジサーバとして Many-core マシンを使用することを提案する．このサーバを利用する際には複数の ROS アプリケーションを効率よく並列実行するための ROS における Many-core プロセッサへのタスク割り当てを行い，その有効性を確認するものとした．

4.5 プロセッサ割り当て手法

アプリケーションへのプロセッサの割り当てにはプロセッサ親和性(Affinity)と呼ばれるスケジューリングを用いた． これにより特定のスレッドの実行を，特定のプロセッサに括り付けることができる．実行を一つのプロセッサに限定することで，あるプロセッサで実行し，中断した後に別のプロセッサで再び実行するという場合に発生するキャッシュ無効化による実行性能の低下を避けることが期待される．

提案システムの概要を図 11 に図示する．本実験で対象とする ROS Node をプロセッサ割り当てするために Node

を初期化した段階でプロセッサ割り当てを行う事とした．割り当て時のアルゴリズムとして今回は Node がランチされた順に連番のプロセッサに割り当てた．ROS は複数のマシン上で別々に Node を実行できることから各マシンに実行している Node を管理するマネージャを実行することとした．このときにマネージャが持つ割り当て管理テーブルを使って Node を管理する．

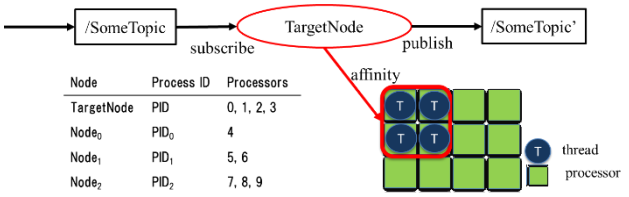

図 11 提案システム概要

5. 評価

5.1 評価 1: コールバック関数計測ツールの確認

5.1.1 評価手法

提案 1 に対して，実際に狙った ROS アプリケーションの実行時間を計測が可能か確認を目的として，以下の評価実験を行った．

はじめに，SLAM 処理を行う slam_gmapping package に提案手法を適用したときの SLAM 処理の実行時間を計測した．実験には 3 次元ロボットシミュレータである Gazebo シミュレータと ROS 対応のロボットの turtlebot3 burger を使用した．実験ではシミュレーション空間上に設置した TurtleBot3 burger からのセンサデータをエッジサーバに送信し，エッジサーバで SLAM の処理を行った．このときに 4.3 章の方法を用いて処理にかかる時間を計測した．

実験環境を次の表 2 に示す．実験は VM 上に用意した環境にて行った．VM ホストには Intel Xeon Gold 6230 (20 core 40 thread) を 2 基搭載しメモリは DDR4 188GiB 搭載する Ubuntu18.04.5 マシンを使用した．ハイパーバイザーには KVM を使用した．VM のゲストにはすべてのプロセッサが利用できるように設定し，メモリは 32GiB とした．また VM ゲストの OS はホストと同じ Ubuntu 18.04.5 を使用，ROS のディストリビューションには melodic を選択した．

実験に使用した Gazebo シミュレータにて使用したワールドは turtlebot3_world である．また，SLAM 処理によって生成された環境地図を rviz を使い表示させた．実験の様子を図 12 に示す．この環境で 10 分間 Gazebo シミュレータと TurtleBot3 を用いた 3 次元 SLAM 処理を行い，時間を計測した．

表 2 評価 1: 実験環境

CPU	Intel Xeon Gold 6230 (20Core, 2.1GHz, 27.5MB cache) x 2
CPU	キャッシュ 27.5MB
VMhostRAM	188GiB
VMhostOS	Ubuntu18.04.5
VMgestRAM	32GiB
ROS dist	melodic

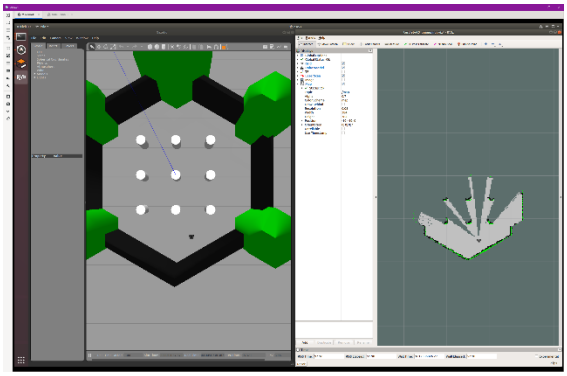

図 12 評価 1: 実験の様子

5.1.2 結果

結果を次図 13 に示す．

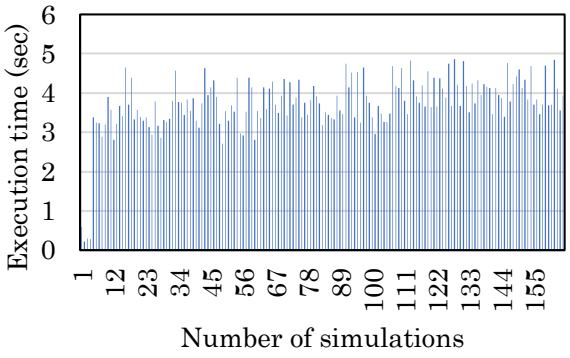

図 13 評価 1: 実験結果

実験の 10 分間のうちに 165 回 SLAM Node が実行された．初回の 4 回を外れ値として扱ったときの SLAM 処理にかかる平均実行時間は 3.81 秒であった．また，最小値は 2.71 秒，最大値は 4.86 秒，分散は 0.25 であった．結果より，コールバック関数の計測が可能となったことがわかった．

5.2 評価 2: SLAM 実行時のプロセスの Many-core 割り当ての性能評価

5.2.1 評価手法

ROS の Node をプロセッサに割り当てて実行することができる提案システムの有効性を調べるため提案手法を適用していない既存の ROS と提案システムとを使い評価を行った．評価では SLAM 処理の実行時間を評価した． ROS Node をプロセッサに割り当てた時の効果を測定するため SLAM 処理の実行時間を計測し比較を行った．実験には複数のロボットが単一のエッジサーバに接続し処理を行いその結果を受け取るという流れで 10 台のロボットからのデ

ータをエッジサーバで受信し，SLAM 処理を行ったのちにその結果を返す．ロボットは ROS のための 3 次元ロボットシミュレータである Gazebo シミュレータ上に 10 台設置し，ここからデータをエッジサーバに集める．ROS では同じ Topic 名で異なるデータを送信できないため 10 台のロボットは異なるネームスペースを使いデータを送信する．

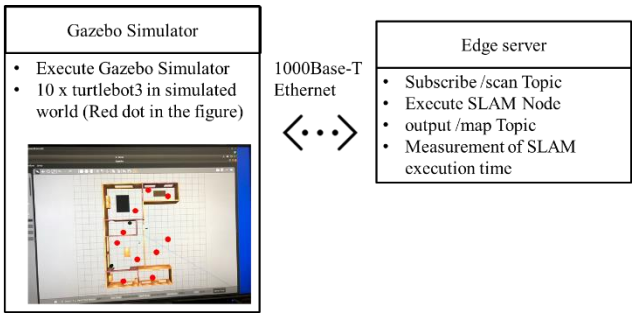

図 14 マシンと実験条件

実験環境を次の表 3 に示す．

表 3 評価 2: 実験環境

Edge Server	CPU	Intel Xeon Gold 6230 (20Core, 2.1GHz, 27.5MB cache) x 2
	CPU	cache27.5MB
	RAM	188GiB
	OS	Ubuntu18.04.5
	ROS distro	melodic
Gazebo Simulator	CPU	Ryzen9 3950X
	GPU	GeForce RTX3080
	RAM	32GiB
	OS	Ubuntu 18.04.5
	ROS distro	melodic

エッジサーバではロボットからのデータをもとに処理を行う事を想定してロボット自身や周辺環境をシミュレーションする Gazebo は同じネットワーク内の別のマシンでホストした．2 台のマシンはスイッチングハブを経由して 1000BASE-T のイーサネットで接続してある．エッジサーバとして Intel Xeon Gold 6230 (20 core 40 thread) を 2 基搭載しメモリは DDR4 188GB 搭載する Ubuntu18.04.5 マシンを使用した．ロボットは TurtleBot3 burger を使用し Gazebo シミュレータの café world 内に 10 台バラバラに設置した．それぞれのロボットから Publish される/scan Topic をエッジサーバに集約する．

5.2.2 評価結果

実験結果を図 15 に示す．TurtleBot3 burger を 10 台使用して実験を行ったが，提案手法あり，なしの場合共にロボット 5 のデータが取れず 9 台分のデータを用いて評価を行った．提案手法なしの場合は実験の 10 分間のうちに 9 台で合計 1162 回 SLAM Node が実行された．初回の外れ値をのぞいたときの SLAM 処理にかかる平均実行時間は 4.72 秒であった．また，最小値は 2.40 秒，最大値は 8.58 秒，分散は 1.42 であった．一方で提案手法を取り入れた ROS では 9 台の合計 1268 回 SLAM Node が実行された．こちらも初回の外れ値を除く SLAM 処理にかかる平均実行時間は 4.35 秒で，1.6%の実行性能向上となった．また，最小値は 2.20 秒，最大値は 12.6 秒，分散は 1.85 となった．

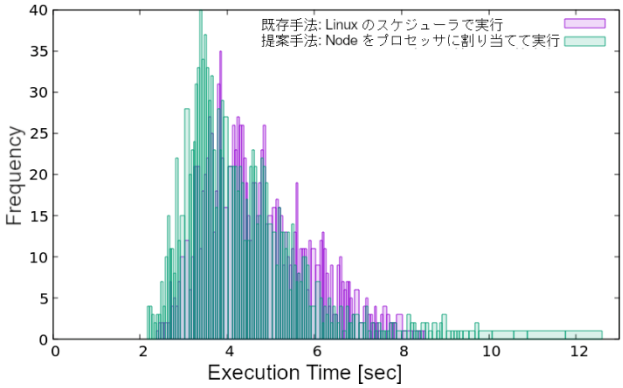

図 15 評価 2: 実行時間の分布

図 15 の結果より，提案手法を取りいれていない ROS と比較したときに平均値で約 1.6%実行時間を短縮することができた．それに伴い 10 分間の間に実行した回数も 96 回増加している．

提案手法を使用しない場合，Node へのプロセッサの割り当てには OS の標準のスケジューラを使用しているためコールバック関数の呼び出し毎に任意のプロセッサで実行されていた．しかし，Node をプロセッサに割り当て，プロセッサの親和性を優先するシステムを使用したためプロセッサのキャッシュが有効に使えることから実行時の効率が向上したといえる．一方で提案手法を適用した場合に既存手法と比べて実行時間が異常な長さになる場合が多い事もわかった．実行回数が増加したことに関してだが，Subscriber では Publisher から送られるメッセージをキューに格納しそこから取り出して処理を行う．そのため 1 回あたりの処理時間を短縮することができたため同じ時間内での実行回数を増やすことができたと考えられる．

本提案手法では性能の向上率がわずかであるという課題がある．この原因として 2 つ考えられる．まず，(1) 実験時にプロセスマイグレーションが頻繁に起こらなかった可能性がある．提案手法では明示的にプロセス(ROS Node)をプロセッサ割り当てすることでキャッシュのコヒーレンシを保つことを考えた．しかし，既存手法である Linux の標準スケジューラを用いて ROS Node を実行したときも基本的には一度実行したときのプロセッサに再び割り当てて実行する．このため，プロセスマイグレーションが起きなければどちらの手法でもキャッシュの一貫性が保たれる．しかし，既存手法ではプロセスマイグレーションが起きた時に

異なるプロセッサで実行される可能性がある．このときにキャッシュのコヒーレンシを保つことができないために同期処理が必要となるため遅延が生じる．本実験では 80 プロセッサが利用可能な条件で 10 ノードの実行をしたためにプロセスマイグレーションが起きにくい状況だったのではないかと考えられる．

性能向上割合が小さくなる 2 つ目の原因として(2)SLAM の実行時間に対してキャッシュのコヒーレンシによる実行時間への影響割合が小さいためであると考えた．

6. まとめ

本論文では複数台のロボットによる SLAM 処理に焦点を当て Many-core エッジサーバで効率よく実行する手段を提案，評価を行った．まず，ROS 上で Node の実行時間を正確に計測するツールを設計実装した．これを用いて，キャッシュを考慮して ROS Node を Many-core プロセッサに割り当てたときの実行時間を計測した.その結果 1.6%の実行性能の向上を実現することができた．しかし，その結果は必ずしも大きな差とはならなかった．プログラムの配置などの検討により改善の余地があると考えられることから，今後引き続き，より有効な割り当てについてのアルゴリズムを検討する．

謝辞
本研究は JST, CREST, JPMJCR19K1 の支援を受けたものです. ここに感謝いたします.

参考文献

[1] "Haneda Robotics", https://tokyo-haneda.com/hanedaroboticslab/, (参照 2021-06-10)
[2] “ 自律走行式 ロボット床面洗浄機「SE-500iX」新発売 | 2014 年 | 製品情報・ニュースリリース | アマノ株式会社”, https://www.amano.co.jp/info_event/20140324_482.html, (参照 2021-06-10)
[3] “SEQSENSE（シークセンス） | Security Robot System”, https://www.seqsense.com,(参照 2021-06-10)
[4] G. Mohanarajah, D. Hunziker, R. D'Andrea and M. Waibel, "Rapyuta: A Cloud Robotics Platform," in IEEE Transactions on Automation Science and Engineering, vol. 12, no. 2, pp. 481-493, April 2015, doi: 10.1109/TASE.2014.2329556.
[5] "AI+IoT 連携型クラウドロボティクスサービス - 株式会社ヘッドウォーターズ", https://www.headwaters.co.jp/service/pepper/synapps.html, (参照 2021-06-19)
[6] “FlytBase”. https://flytbase.com/ (accessed 2020-09-29)
[7] "フィールドロボットの 現状と課題 - NEDO", https://www.nedo.go.jp/content/100563899.pdf, (参照 2021-06-12)
[8] Kashif Bilal, Osman Khalid, Aiman Erbad, Samee U. Khan, “Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers”, Computer Networks, Vol. 130, pp. 94-120, Jan. 2018.
[9] “ROS/Technical Overview - ROS Wiki “ http://wiki.ros.org/ROS/Technical%20Overview，(参照 2021-06-19)
[10] L. Yang and H. Chi, "SLAM Self - Cruise Vehicle Based on ROS Platform," 2021 IEEE International Conference on Consumer Electronics and Computer Engineering (ICCECE), pp. 6-11, 2021
[11] “ROS”, https://www.ros.org，(参照 2021-06-12)
[12] “rqt_graph - ROS Wiki”, http://wiki.ros.org/rqt_graph, (参照 2021-06-19)
[13] "rosbag - ROS Wiki", https://wiki.ros.org/rosbag，(参照 2021-06-14)
[14] 谷本輝夫, 佐々木広,三輪忍,中村宏.メニーコアプロセッサにおける競合とスケーラビリティを考慮したスレッドスケジューリング.研究報告ハイパフォーマンスコンピューティング（HPC）.31 号. 2011-11-21
[15] 三好健文，笹田耕一，植原昂，佐野伸太郎，森洋介，吉瀬謙二.メニーコア向けタスクスケジューリングシステムの検討．研究報告計算機アーキテクチャ（ARC）．10 号．2009-078-26