Autoware on Many-core Platform: NoCベース組込みメニーコアプロセッサ向け自動運転プラットフォーム

安積 卓也[1]　丸山 雄也[2]　前田 青也[1]　加藤 真平[3]

概要：近年，自動運転を始めとする様々な組込みシステムにおいて，低消費電力での高い計算性能が求められている．メニーコアプロセッサはそれらの要件を満たすことから，実用化に向けた様々な研究が進められているが，大規模・複雑化する組込みシステムに対応するため，メニーコア向けソフトウェア開発の生産性向上が要求されている．本研究では，NoC をベースとしたメニーコアプロセッサを用いた自動運転プラットフォーム (Autoware) の並列化を行い，構造化されたメッセージ通信を提供するミドルウェア及びそのソフトウェア開発フレームワークを提案する．性能評価では，低速自動運転で求められているデッドラインを満たすことを示す．

キーワード：メニーコアプロセッサ，自動運転プラットフォーム，ROS（Robot Operating System）

Autoware on Many-core Platform: Self-driving Platform for NoC-based Embedded Many-Core Processor

Takuya Azumi[1]　Yuya Maruyama[2]　Seiya Maeda[1]　Shinpei Kato[3]

Abstract:
In resent years, embedded systems including self-driving systems often require high processing capability and low power consumption. Since many-core processors meet the requriments, software for many-core platforms must be designed based on scalable data allocation and scalable parallelization. This paper proposes a parallel computing and software development framework for embedded many-core platforms based on network-on-chip (NoC) technology. To demonstrate the practicality of embedded many-core platforms in a practical application, a module of a self-driving software platfrom (Autoware) is parallelized so that it can run on many-core processors. The experimental results show the the proposed framework and the parallelized applications meet the deadline of low-speed self-driving systems.

Keywords: Many-core processor, Self-driving Platform, ROS (Robot Operating System)

1. はじめに

近年，自動運転システムをはじめとする組込みシステムの領域では，高い処理要求と低電力消費を同時に実現するため，マルチ/メニーコアに向けたコンピューティングプラットフォームの進化が求められている [2] [6] [14]．例えば，自動運転システムは LiDAR やカメラから得られた情報をもとに，自己位置推定，環境認識，経路経過，経路追従等様々な処理を同時に行う必要がある．自動運転システムの高い処理要求，予測可能性，エネルギー効率の要件を考慮すると，マルチ/メニーコアや GPU 等のヘテロジニアスなコンピューティングシステムが必要である．

しかし，マルチ/メニーコアプラットフォームを組込みシステムに適用するにはいくつかの問題が存在する [2] [16]．例えば，並列化されたプロセス同士で頻繁にアクセス競合が発生する問題や，共有資源が予測可能なタイミング動作やソフトウェア分析を妨げること等が挙げられる．これらの問題は，組込みシステム特有の厳格な要求仕様や多くのコアによって資源（例えば，メモリ及び I/O デバイス）を共有することに起因している．さらに，大容量メモリと全てのコアを広帯域バスで接続してしまうことで，バス競合によってコア数の拡張性が失われ，大きな電力消費が必要

[1] 埼玉大学
Saitama University
[2] 大阪大学
Osaka University
[3] 東京大学
The University of Tokyo

なってしまう問題も存在する．このコア数の拡張や消費電力の問題に対応するために，Kalray によって開発された Multi-Purpose Processing Array (MPPA) -256[12] では，1 クラスタ 16 コアの 16 クラスタ（256 コア）を採用し，クラスタ間をネットワークオンチップ（NoC）でつなぐ構成をしている．

組込み要件を考慮し，コアの数の拡張と妥当な電力効率のために，NoC を用いた分散メモリアーキテクチャを採用している．本研究では，組込み向け NoC ベースメニーコアプラットフォームの実アプリケーションへの適用を取り上げる．

貢献：本研究では，MPPA-256 等の NoC に基づく組込みのメニーコアコンピューティングの検証に重点を置く．主に，自動運転のレベル 4（バス等経路が決まっている経路を走る自動運転のレベル）を対象とした自動運転プラットフォームを動作させることで，本研究の有用性を示す．

- 自動運転処理で特に重い処理である自己位置推定をメニーコアにオフロードすることで，LiDAR のデッドラインである 100ms 以下に処理を終了させる
- メニーコア向けの ROS(ROS-lite) を提案し，経路計画や経路追従のノードをメニーコア上で実行する

構成：本論文は，以下のように構成される．まず，本論文で想定するシステムモデルについて 2 章で説明する．ここでは，Kalray MPPA-256 Bostan のハードウェアモデルとソフトウェアモデルを提示する．3 章では，提案フレームワークを説明する．次に，4 章では，評価実験の構成やアプローチについて説明し，評価内容の考察を行う．続いて，5 章では，マルチ/メニーコアシステムに焦点を当てた関連研究について議論する．最後に，6 章にて，まとめとして今後の課題と結論を示す．

2. システムモデル

本章では，評価に使用される Kalray MPPA-256 Bostan のシステムモデルを示す．まず，ハードウェアモデルを紹介し，その後，ソフトウェアモデルを説明する．

2.1 ハードウェアモデル

MPPA-256 プロセッサは，16 個のコンピュートクラスタ（CC）及び 4 個の I/O サブシステム（IOS）で構成され，これらは全てトーラス状のネットワークオンチップ（NoC）によって接続されている（図 1 と図 2 参照）．

2.1.1 I/O サブシステム（IOS）

MPPA-256 には，北，南，東，西の 4 つの I/O サブシステム（IOS）がある．北と南の IOS は，DDR インタフェースと 8 レーンの PCIe コントローラに接続されている．東と西の IOS は，10GB/s イーサネットコントローラに接続されている．各 IOS は 4 つの I/O コアと NoC インタフェースで構成されており，図 1 のように 2 つの IOS によって 1 つの I/O クラスタ（IOC）が構成される．

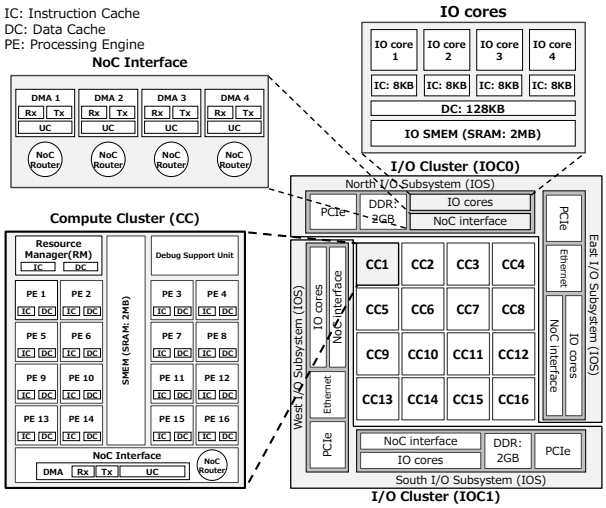

図 1　Kalray MPPA-256 Bostan のアーキテクチャの概要

Fig. 1　An overview of the architecture of the Kalray MPPA-256 Bostan.

I/O コア：I/O コアは，図 1 に示すように，合計容量が 2MB の共有メモリ（IO SMEM）に接続される．各 I/O コアは 32（8×4）KB の独自の命令キャッシュを持ち，128KB のデータキャッシュ及び外部の DDR メモリへのアクセスを 4 つの I/O コアで共有する．さらに，I/O コアは，PCIe，Ethernet，及びその他の I/O デバイスコントローラ等を管理している．

2.1.2 コンピュートクラスタ（CC）

MPPA-256 では，16 個のコアを持つコンピュートクラスタ（CC）が 16 個存在し，合計 256 のコアによって主要な処理が行われる．NoC 上の 16 個の内部ノードが CC に対応しており，図 1 は各 CC のアーキテクチャを示している．

プロセッシングエンジンとリソースマネージャ：CC では，16 個のプロセッシングエンジン（PE）と 1 個のリソースマネージャ（RM）が 2 MB のクラスタローカルメモリ（SMEM）を共有する．PE は主に，並列処理のためにユーザによって使用される．CC 内の PE と RM は，Kalray 独自の 32 ビット VLIW の Kalray-1 コア (600MHz または 800MHz) となっている．

2.1.3 ネットワークオンチップ（NoC）

16 個の CC と 4 つの IOS は，図 2 のようにネットワークによって接続されている．プロセッサ上にバスネットワークとして NoC が構築されており，各ノードにはルータが存在する．

バスネットワーク：バスネットワークはノード（CC と IOS）をトーラス状に接続している [12]．トーラス状ネットワークのメリットはメッシュ状 [17] の場合と比べて平均ホップ数が少なくなることにある．実際はネットワークは双方向リンク（図 2 に赤線で示される）を持つ 2 つの並列バスで構成されている．大きな容量のデータ転送に最適化された Data-NoC（D-NoC）と低レイテンシで小さなメッセージ用に最適化された Control-NoC（C-NoC）である．

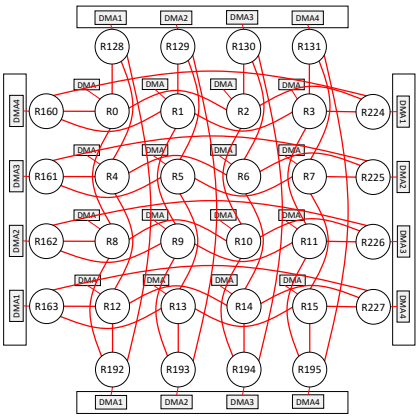

図 2　NoC（Network-on-Chip）マップ
Fig. 2　NoC（Network-on-Chip）map.

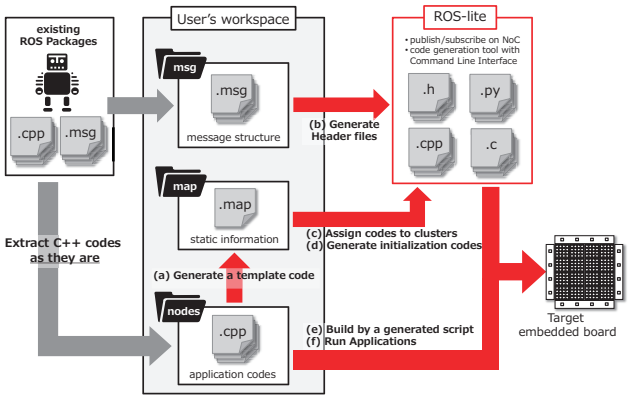

図 3　ROS-lite フレームワークの開発の流れ
Fig. 3　Development flow of ROS-lite framework.

NoC ルータ：CC 毎の 1 つノードと IOS 毎の 4 つのノードには，D-NoC ルータと C-NoC ルータがある．さらに，CC/IOS 上の NoC インタフェースの DMA エンジンは，Rx インタフェース，Tx インタフェース，及び UC を備えた D-NoC ルータを介して送受信を行う．図 1 に示されている NoC ルータは，図 2 のノードと対応する．D-NoC と C-NoC の両方で，各ネットワークノード（CC または IOS）には次の 5 つの方向へのリンクを持っている．東西南北の 4 つの隣接リンク及び，及び NoC ルータに接続されたローカルアドレススペースへのリンクの計 5 つである．NoC ルータは，各方向でフリットを FIFO キューイングし，リンク幅は各方向 4 バイト幅であり，コアが 600MHz または 800MHz の CPU クロックレートで動作するため，各ノードは合計 2.4GB/秒または 3.2GB/秒で送受信できる．

2.2　ソフトウェアモデル

本論文で用いるリアルタイム OS である eMCOS について説明する．eMCOS は，組込みシステム向けメニーコアリアルタイム OS であり，CC と IOS の両方で動作し，最小限のプログラミングインタフェースとライブラリを提供する．eMCOS は分散マイクロカーネルアーキテクチャを実装しており，これにより，アプリケーションは優先順位ベースのメッセージパッシング，ローカルスレッドスケジューリング，スレッド管理を IOS 及び CC で操作することが可能になっている．

- **通常メッセージ**：eMCOS は D-NoC の Tx インタフェース用いたスレッド間の通常メッセージ API を提供している．メッセージの送信先はスレッドで，送受信は，非同期・同期モードを利用できる．eMCOS は IOS 及び CC で，全てのクラスタ間で同様に動作させることができる．各コアのメッセージバファは eMCOS で管理され，データは D-NoC を経由して DMA で送信される．受信順序が確定すると，ユーザ空間のメッセージバッファにデータがコピーされる．
- **セッションメッセージ**：eMCOS は D-NoC を用いた UC インタフェースを利用した大容量の転送を想定したセッションメッセージ API を提供している．この API は，自動運転の 3D マップや経路情報等大きいサイズデータを繰り返して送る場合に向いている．利用者は，送受信それぞれのペアに関連したセッションを作成する必要がある．セッションの数の上限は，各クラスタの UC の数に依存する．予めセッションを確立しておくことで，クラスタ間でデータのやり取りができるようになる．

3.　ROS-lite

提案フレームワーク (ROS-lite) は，メニーコアプラットフォーム上での効率的なアプリケーション開発を実現する通信レイヤを提供する．ROS や関連研究 [11] [10] では，メニーコアの通信で用いられる NoC 通信を提供しておらず，リソース制約の厳しい組込みシステムでの利用は難しい．ROS-lite は，これらの問題を解決し，ROS ノードをメニーコアの各コアで実行し，コア間もしくはクラスタ間での通信を実現するための開発環境を提供する．ROS-lite は，できる限り既存の ROS ノードのコードを適用するための機能も提供する．さらに，ROS-lite は既存トピック通信に準拠しており，既存のプラットフォームで実行される ROS ノードと通信できる機能も提供する．本論文では，eMCOS をリアルタイム OS として，MPPA-256 を組込みメニーコアプラットフォームとして用いる．

3.1　ROS-lite ツールチェーン

本節では，ROS-lite の開発の流れについて説明する．図 3 に，ROS-lite の開発の流れを示す．ROS-lite は ROS ノードと同様のアプリケーションコードを実行できる．ROS と同様に C++で実装することを想定している．アプリケーション開発者は，既存の ROS アプリケーションコードとメッセージファイルを変更せず ROS-lite のアプリケーションコードとして適用できる[*1]．タスクマッピングに関して，開発者はマップファイル (`.map`) を作成することによって，

[*1]　特殊なライブラリを利用している場合は別途移植は必要になる．

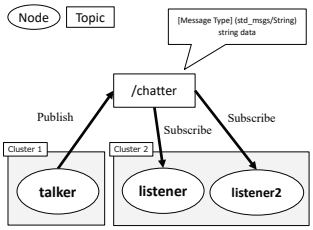

図 4　ROS-lite における publish/subscribe モデル
Fig. 4　The publish/subscribe model in ROS-lite.

```
1:- name: talker
2:  cluster: 1
3:  publish: [/chatter]
4:  subscribe: []
5:- name: listener
6:  cluster : 2
7:  publish: []
8:  subscribe: [/chatter]
9:- name: listener2
10:  cluster : 2
11:  publish: []
12:  subscribe: [/chatter]
```

図 5　マップファイル (.map).
Fig. 5　Map file (.map).

ROS ノードをメニーコアプラットフォームの CC 上及びコアにマッピングできる．開発者は，ソースコードを修正せずに，マップファイルを修正するだけでタスクマッピングを変更できる．ROS ノードは，リアルタイム OS が提供するスレッドとして実現しており，publish/subscribe モデル，メッセージファイル (.msg) で定義された構造を扱うことができる．

ROS ノード間の通信は，クラスタ間・クラスタ内のそれぞれの通信を提供している．クラスタ間の通信は，NoC 通信で，クラスタ内の通信は共有メモリを利用して実現する．

ROS-lite のツールチェーンは，効率的な開発を実現するためにコマンドラインでのコード生成及び，ビルドシステムを提供している．図 3 (a) では，既存の ROS のソースコードからマップファイル (.map) のテンプレートコードを自動生成する．マップファイルは，ノード名，所属クラスタ名，パブリッシュもしくはサブスクライブするトピック名の情報を含む．図 5 にマップファイルの例を示す．マップファイルの所属クラスタ以外の情報は，ROS ノードのソースコードから情報を抜き出し，マップファイルのテンプレートを生成する．アプリケーション開発者は，テンプレートファイルに所属クラスタ番号を追加するだけで良い．

次に，図 3 (b) に示すとおり，ROS と同様にメッセージファイル (.msg) から，メッセージ構造体を含むヘッダファイルを自動生成する．ROS-lite で必要な情報のみを自動生成するため，生成されたヘッダは ROS のヘッダより軽量になる．この自動生成スクリプトは，オリジナルの ROS のスクリプトをベースに開発している．メッセージファイルは ROS と同様の表現ができ，ROS のメッセージファイルを修正する必要はない．

図 3 (c), (d) に示すとおり，ROS ノードの実行のための初期化コードを自動生成する．これらのコードはマップファイルの情報を利用して生成する．ROS ノードは，開発者によって割り当てられたクラスタのリアルタイム OS によって，スレッドによって実行される．

図 3 で示すとおり，ビルドスクリプトは，マップファイルの情報から自動生成される．各クラスタでメモリバングが分けられているため，ROS ノードのソースコードは，各クラスタ毎にビルドされる．これらのビルドは生成されたビルドスクリプトによって実行される．マップファイルを修正によるタスクマッピングの変更が行われた場合に，ビルドスクリプトは，生成し直される．ROS のコードを同じディレクトリに保存しておき，各ノードは，各クラスタ毎に自動的にビルドされる．

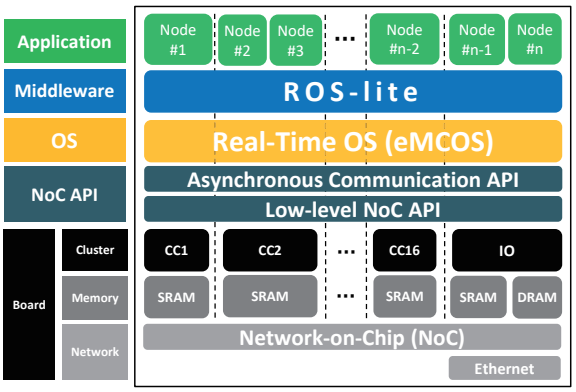

図 6　メニーコアプラットフォーム上の ROS-lite システムスタック
Fig. 6　System stack of ROS-lite on a many-core platform.

ROS-lite フレームワークを理解するために，一つのパブリッシャと 2 つのサブスクライバ及び *chatter* トピックの例（図 4）で説明する．*chatter* トピックは，文字列型の data で定義されている．図 5 のマップファイルで定義されているとおり，パブリッシャノードはクラスタ 1 に 2 つのサブスクライバはクラスタ 2 で実行される．開発者は，マップファイルのクラスタ番号を変更だけで，ROS ノードのマッピングを変更できる．マップファイルに定義されたトピック情報は，ROS ノードとトピックの初期化で利用され，ROS-lite のノードとトピックの関係を静的に生成できる．ノード名とトピック上の項目は，ROS のソースコードから自動生成され，ノードの関係を初期化するコードは，マップファイルの情報から自動生成される．ただし，クラスタ番号の割当ては開発者によって行う必要がある．

3.2　設計と実装

ROS-lite は，図 6 に示す MPPA-256 上で実装している．MPPA-256 は，2.1 節で説明したとおり，分散メモリアーキテクチャの NoC ベースのメニーコアプロセッサである．ROS-lite は，省メモリの NoC ベースのメッセージ通信を実現しており，組込みアプリケーションと同様にメモリ制約のあるメニーコアプラットフォーム上で実行できる．ROS-lite は MPPA-256 に依存した設計にはなっておらず，他のハードウェアプラットフォームにも適用可能[*2]である．

2.1.3 節で説明したとおり様々なメッセージ API を提供しており，CC 及び IOS の両方で実行できるため，MPPA-256

[*2] 参照実装として，MPPA-256 以外に Jetson TX2 で実装したコードをオープンソースで公開 (https://github.com/azu-lab/ros-lite) している．

向けの ROS-lite の実装に eMCOS をリアルタイム OS として利用している．ROS-lite は特定の OS に依存しない設計になっている*3全ての ROS-lite のスレッド管理は eMCOS を利用して行っている．

ROS-lite は，組込みシステム向けにノードを静的に構成する設計にしているため，ノード結合を管理する ROS マスタは必要がない．マスタが行っているノード間の結合は，マップファイルから自動生成された初期化コードで行っている．初期化コードは，ROS-lite のコマンドラインでコンパイル前に実行され生成される．ROS ノードの結合処理等の初期化処理は，ROS マスタの排除により，不必要なクラスタ間の通信処理をなくすことができ，ROS で問題となっているマスタによる単一点障害*4も起きない．

次に，ROS-lite における publish/subscribe モデルの内部実装について述べる．サブスクライバのメインスレッドによって，各トピックに対してメッセージ受信関数が生成される（図 7）．サブスクライバノードは，メッセージをバッファリングし，トピック受信時に ROS-lite のコールバック処理を行う．図 7 に示す設計は，2 章で説明した eMOCS のメッセージを利用することで実現できる*5．メッセージ関数は，メッセージバッファやスレッド間 NoC 転送を提供している．しかし，バッファはカーネル領域で確保され，動的に確保できない．そのため，経路情報や 3 次元地図等可変長のデータのやり取りに適していない．ROS-lite はセッションメッセージを利用することでこの問題を解決している．図 8 は，ROS-lite での publish/subscribe モデルの実装を示している．パブリッシャノードは，データ転送の前にトピックのデータ構造に従ってデータをシリアライズする．シリアライズの方式は，ROS との通信することを考慮し，元の ROS と同等のシリアライズ方式を採用している．シリアライズ後，パブリッシャノードは，マップファイルのトピック情報から静的に生成されたリストにあるサブスクライバノードにメッセージを送る．シリアライズデータサイズを受信したサブスクライバノードは，メモリを確保しデータ転送用のセグメントを作成する．クラスタ間の NoC 通信のセグメントは，通信の度に作成され破棄される．通信によるオーバヘッドはあるが，この方式を採用することで，ROS-lite は，組込みシステムの限られたメモリを有効活用ができ，可変長のメッセージを柔軟に扱えるようになる．なお動的メモリ確保には，リアルタイムメモリアロケータである TLSF[13] を利用している．

コールバック関数の同時実行を避けるには，図 7 の代わりに，図 9 の設計を用いる．コールバックキューは，サブスクライバノードのコールバック関数の実行順序を管理する．コールバックキューは，eMCOS のメッセージバッファを用いることで実現できる．eMCOS メッセージは，サイズの小さなデータ転送に適している．キューに追加されるデータは，コールバックのポインタ及び受信メッセージのポインタであり，キューが消費するメモリサイズが小さい．図 11 に， 図 9 の設計の実装例を示す．こちらの設計の場合は，コールバックキューで管理できるため，コールバック毎にスレッドを生成しない設計も可能となる*6．

ROS-lite は元の ROS アプリケーションコードの移植をサポートし，メニーコアプラットフォームでの効率的な開発を実現する．ROS-lite は，NoC インタフェースに慣れていない開発者でも利用できるようにオリジナルの ROS と同じ API を提供している．図 4 に示す taker/listener のサンプルコードは，ROS のコードと同じものが利用できる．

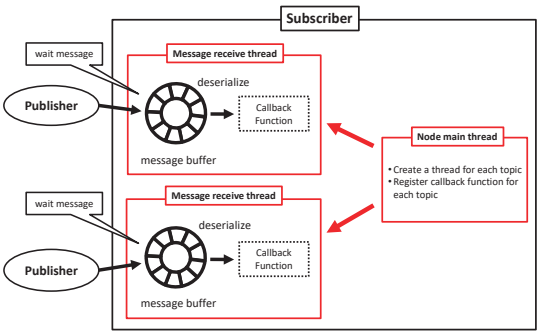

図 7　サブスクライバノードの構造
Fig. 7　Structure of subscriber node.

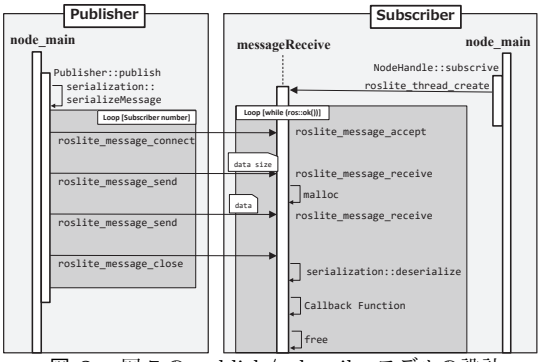

図 8　図 7 の publish/subscribe モデルの設計
Fig. 8　Design of publish/subscribe model of Figure 7.

3.3　事例：自動運転アプリケーションの実行

メニーコア上で動作させる自動運転プラットフォームのノード構成を図 12 の右側に示す．メニーコア上で動作させるアプリケーションとして，レベル 4 で利用される自己位置推定，経路計画及び，経路追従を選択した．物体認識に関しては，データ並列を実現する GPU または FPGA 等で処理させることが想定されているため，今回は対象外とした．その他，LiDAR 等のデバイスドライバ等もメニーコア外で処理することを想定している．自己位置推定

*3　Jetson 向けには，POSIX 系 OS で動作確認できている．
*4　ROS マスタが動作できなくなると，ノードの結合・削除等ができなくなり，システム全体が動作できなくなる問題である．ROS2 では DDS（Data Distribution Service）を導入することでこの単一点障害を防いでいる．
*5　Jetson 版では，UNIX ドメインソケットで実装している．
*6　本論文の事例では，コア数に余裕があるため各トピック毎にスレッドを作成して利用している

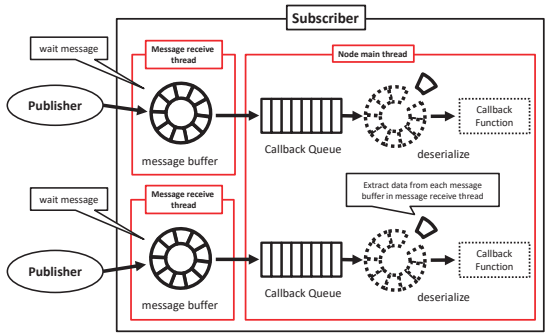

図 9　コールバックありのサブスクライバノードの構成

図 10　Structure of subscriber node with callback queue.

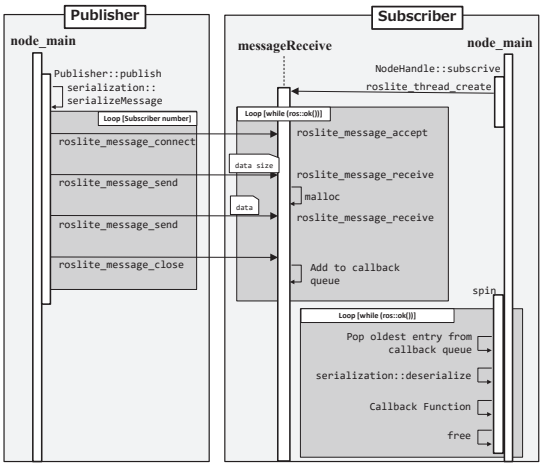

図 11　図 9 の publish/subscribe モデルの設計

Fig. 11　Design of publish/subscribe model of Figure 9.

（lidar_localizer）は，LiDAR から得られた点群と点群の 3 次元マップとマッチングするアプリケーションで，自動運転で最も処理が重いアプリケーションの一つである．

図 12 に示すとおり，これらのノード構成は，バス等予めルートが決まっているレベル 3 もしくは 4 の低速自動運転で利用されるノード構成である．オープンソースの自動運転プラットフォームである Autoware[1] のバージョン 1.8 を用いている．

メニーコアプロセッサを用いた事例として，オフロードと ROS-lite の 2 つに分けている．オフロードのノードとして，自己位置推定（lidar_localizer）を用いた．

自己位置の推定のアルゴリズムは，分析の結果，図 13 に示すとおり LiDAR のスキャンデータが 3D マップ（マップを KD 木に変換したデータ）のどの辺りに属するかを検索する *VoxelGridCovariance::radiusSearch* 及び，スキャンデータとマップの適合スコアを求める *NormalDistributionsTransform::computeDerivatives* がボトルネックになっていることが分かった．

本事例では，CC と IOS 間のオフロードには D-NoC 通信を利用し，*radiusSearch* を 2CC 上に，*computeDerivatives* を 5 CC にオフロードし，自己位置推定の並列化を行った．既存研究では，CC のメモリ制限のため *computeDerivatives* のみを CC にオフロードしていたが，本研究では，3D マップの IOS にある DDR から必要なマップを CC の SRAM にスワップする機構を追加することで *RadiusSearch* のオフロードを実現している．

ROS-lite のノードとして経路計画関係のノード（lane_rule，lane_select，waypoint_filter[*7] 及び，velocity_set）経路追従のノードとして（pure_pursuit と twist_filter）を用いた．

4. 評価

本章では，メニーコアプロセッサのオフロード処理及び提案プラットフォームの評価を行う．測定環境は，2 章で示した Kalray MPPA-256 Bostan を利用した．

図 14 の左側グラフは IOS のみで図 12 のノードを実行した場合（並列化前）の結果で，右が並列化（オフロード（データ並列）及び各ノードを並列実行）した結果である．並列化前は 1000ms 程度かかっていた処理が，並列化後は 50ms 程度に処理時間が抑えられており，end-to-end の実行時間（LiDAR のデータ受信後，ステアリング角度，速度の結果を計算するまでの時間を測定した結果）は，デッドラインとして用いられる LiDAR の周期 10 Hz（100 ms）以下で処理が完了していることが分かる．この結果により，提案プラットフォームは，低速自動運転の実車実験での要件を満たしている．提案プラットフォームを用いた実車実験の様子は，こちらで確認できる[*8]．

3.3 節の事例で使用したノードの実行時間を eMCOS の通常メッセージ及び，セッションメッセージで測定を行った．図 16 に示すとおり，全てのノードでの実行時間がセッションメッセージの方が実行時間が短いか同程度である．この結果の違いは，セッションメッセージは NoC 転送に UC を用いているため，転送時間に違いが出たと推測される．さらに，セッションメッセージでは，処理時間の安定していることが分かる．

今回の事例で用いたノードは，Autoware やデータを一部修正する必要があった．例えば，waypoint の情報（経路情報）や高精度 3 次元地図の情報量を一部削減している．CC の SRAM と IOS の DRAM でスワップアルゴリズムを提供するか，SRAM の容量を改善する必要があることを示している．実用的なアプリケーションの観点では，自己位置推定アルゴリズムの並列化及び，経路計画，経路追従を提案フレームワークでの並列実行することによって，自動運転の実環境で NoC ベースのメニーコアプラットフォームが適用可能であることを示せた．

今回の事例では，低速自動運転を実現する最低限のノードを動作させた．そのため，コア数に余裕があり，基本ノード 1 クラスタの割当てにしている．各トピックを処理するためのスレッドを 1 コアに割り当てている．

図 16 の結果が示すとおり，多くの場合でセッションメッセージが eMCOS 通常メッセージより性能が良いことが分

[*7] obstacle_avoid の一部の機能のみを移植した．
[*8] https://youtu.be/4T0Nqx5M1aA

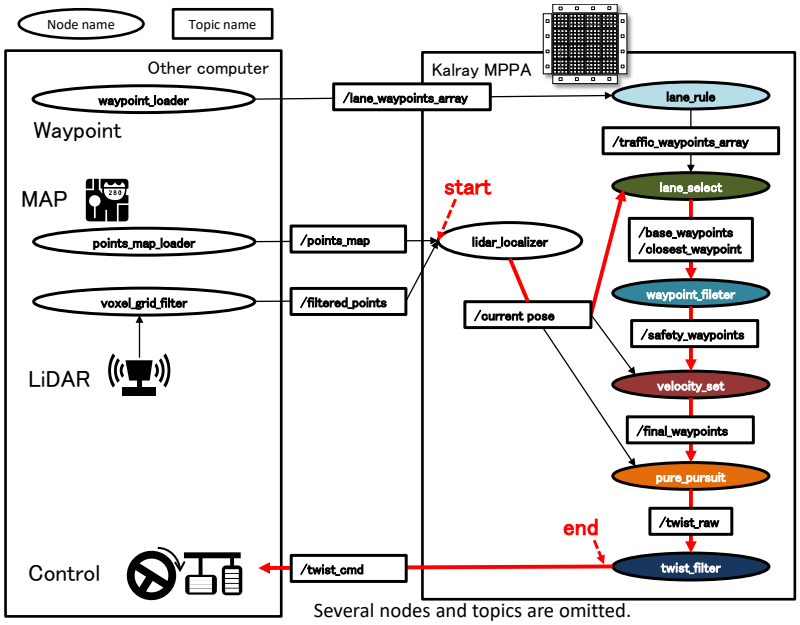

図 12　事例の自動運転プラットフォームのノード構成
Fig. 12　Self-driving nodes for the case study.

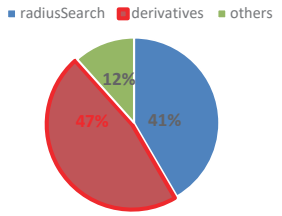

図 13　lidar_localizer のボトルネック
Fig. 13　Bottle neck of lidar_localizer.

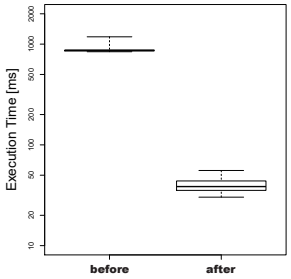

図 14　実行時間の結果
Fig. 14　End-to-end result.

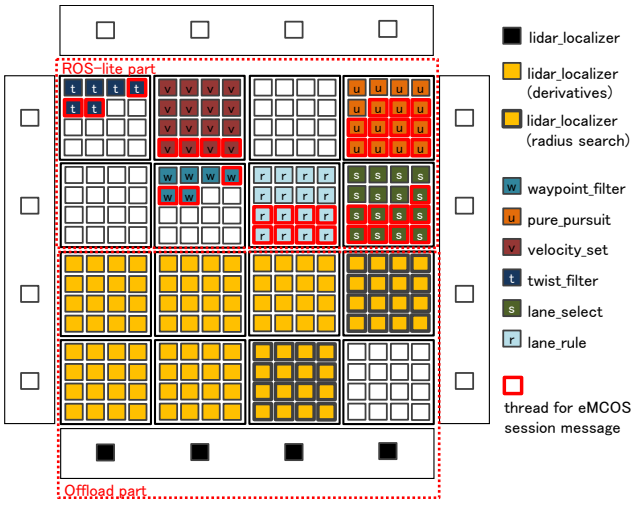

図 15　自動運転プラットフォームのノードマッピング
Fig. 15　A mapping of self-driving platform.

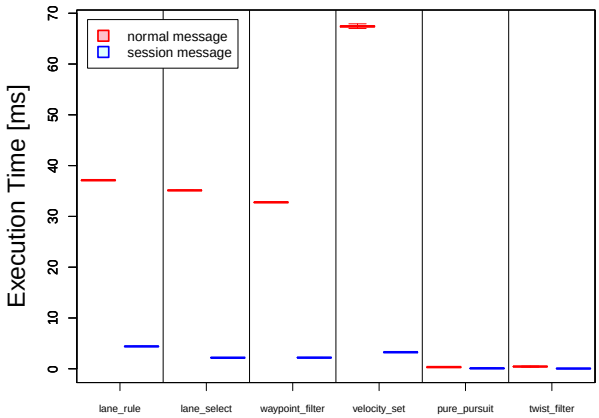

図 16　ノードの実行時間
Fig. 16　Execution time of nodes.

かる．しかし，セッションメッセージは，セッション毎にスレッドが必要であり，通常メッセージより多くのスレッドが必要になる．図 15 中の赤枠は，セッションメッセージで利用するスレッドを示している．図 16 中の pure_pursuit や twist_filter のように，メッセージサイズが固定でサイズが小さい通信に関しては，通常メッセージとセッションメッセージ差は少ない．そのため，トピックの特性や，コアの余裕があるかで通信種類の使い分けを行うと良いことが分かる．

5. 関連研究

既存研究では，MPPA-256 等のメニーコアプラットフォームでのリアルタイムアプリケーションの研究が行われてきた．参考文献 [16] では，マルチ/メニーコアプラットフォームの可能性と今後の課題について述べており，リアルタイム/組込みシステムにおけるマルチ/メニーコアへの移行について議論している．参考文献 [4] [6] [3] では，マルチ/メニーコアシステムのためのタスクマッピングアルゴリズムが提案されており，Airbus は，MPPA-256 を使用したハードリアルタイムアプリケーションのための有向非循環グラフ（DAG）スケジューリングの方法を提案している [3]．参考文献 [6] は，AUTOSAR（車載用組込みソフトウェアシステムを開発するための標準アーキテクチャ [7]）に基づいたマッピングフレームワークを提案しており，共有リソースの競合を考慮した AUTOSAR タスクスケジューリングについては，参考文献 [2] の中で紹介されている．

MPPA-256 を対象にした既存研究は主に参考文献 [5] [8] [15] 等が挙げられる（表 1）．MPPA-256 の性能やエネルギー効率の高さは，参考文献 [8] の中で紹介されている．しかし，このレポートには評価がほとんど含まれておらず，

表 1　関連研究との比較
Table 1　Comparison of Previous Work

	performance analysis	data transfer analysis with NoC	real applications	parallel data transfer
Kalray clusters calculate quickly [8]	L			
Network-on-Chip Service Guarantees [5]		✓		
Predictable composition of memory accesses [15]		✓		
this paper	✓	✓	✓	✓

NoC によるデータ転送やメモリアクセス特性については触れられていない．MPPA-256 の NoC によるデータ転送や NoC の保証するサービスについては参考文献，[5] の中で分析されている．これらの研究では，理論的な分析は十分に行われているが，評価実験等は少なく，データの並列送信は取り上げられていない．参考文献 [15] は，予測可能なメモリアクセスに焦点を当てている．研究では，MPPA-256 等のメニープラットフォームでのパフォーマンスと予測可能性の主なボトルネックとして，外部 DDR との NoC 通信が挙げている．この分析では，外部 DDR へのメモリアクセス特性が取り上げられいくつかの考察を得ているが，ボトルネックに対する解決策は検討されておらず，実用的な評価は欠けている．

一方, ROS のプラットフォームとして ROS2[11] や micro-ROS[10] 等組込みシステムに適した ROS として提案がされているが，メニーコアプロセッサに対応したプラットフォームの提案はされていない．

6. まとめ

本研究では，組込みシステム向けメニーコアプロセッサである MPPA256 を用いた自動運転向けプラットフォームを提案した．大容量データの計算はオフロードとして，自動運転機能は ROS ノードの実行環境としてメニーコアプロセッサを用いた．実車実験では，提案プラットフォームを用いることで，自動運転で必須機能である自己位置推定，経路計画，経路追従を動作を既存プラットフォーム [9] より消費電力の少ないプラットフォームで動作することを確認した．今後は，ROS2 の設計で議論されている機能安全，CI（Continuous Integration）や，軽量 DDS である DDS-XRCE 等を考慮した設計も検討する．

謝辞

本研究は，JST さきがけ JPMJPR1751，国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務及びイーソルの支援から得られたものである．

参考文献

[1] Autoware Foundation: Autoware, https://gitlab.com/autowarefoundation/autoware.ai.
[2] Becker, M., Dasari, D., Nicolic, B., Akesson, B., Nolte, T. et al.: Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform, *in Proc. of ECRTS*, pp. 14–24 (2016).
[3] Becker, M., Mubeen, S., Dasari, D., Behnam, M. and Nolte, T.: Scheduling multi-rate real-time applications on clustered many-core architectures with memory constraints, *in Proc. of ASP-DAC*, pp. 560–567 (2018).
[4] Carle, T., Djemal, M., Potop-Butucaru, D., De Simone, R. and Zhang, Z.: Static mapping of real-time applications onto massively parallel processor arrays, *in Proc. of ACSD*, pp. 112–121 (2014).
[5] de Dinechin, B. D. and Graillat, A.: Network-on-Chip Service Guarantees on the Kalray MPPA-256 Bostan Processor, *in Proc. of AISTECS* (2017).
[6] Faragardi, H. R., Lisper, B., Sandström, K. and Nolte, T.: A communication-aware solution framework for mapping AUTOSAR runnables on multi-core systems, *in Proc. of ETFA*, pp. 1–9 (2014).
[7] Fürst, S., Mössinger, J., Bunzel, S., Weber, T., Kirschke-Biller, F., Heitkämper, P., Kinkelin, G., Nishikawa, K. and Lange, K.: AUTOSAR–A Worldwide Standard is on the Road, *in Proc. of ELIV*, Vol. 62 (2009).
[8] Kanter, D. and Gwennap, L.: Kalray clusters calculate quickly, *Microprocessor Report* (2015).
[9] Kato, S., Tokunaga, S., Maruyama, Y., Maeda, S., Hirabayashi, M., Kitsukawa, Y., Monrroy, A., Ando, T., Fujii, Y. and Azumi, T.: Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems, *in Proc. of ICCPS* (2018).
[10] Lange, R.: Bringing the Next Generation Robot Operating System on Deeply Embedded Autonomous Platform, *in Proc. of ASD* (2019).
[11] Maruyama, Y., Kato, S. and Azumi, T.: Exploring the Performance of ROS2, *in Proc. of the 13th ACM EMSOFT*, pp. 5:1–5:10 (2016).
[12] Maruyama, Y., Kato, S. and Azumi, T.: Exploring Scalable Data Allocation and Parallel Computing on NoC-based Embedded Many Cores, *in Proc. of ICCD* (2017).
[13] Masmano, M., Ripol, I., Crespo, A. and Real, J.: TLSF: a new dynamic memory allocator for real-time systems, *in Proc. of ECRTS*, pp. 79–88 (2004).
[14] Perret, Q., Maurère, P., Noulard, É., Pagetti, C., Sainrat, P. and Triquet, B.: Mapping hard real-time applications on many-core processors, *in Proc. of the ACM 24th RTNS*, pp. 235–244 (2016).
[15] Perret, Q., Maurère, P., Noulard, E., Pagetti, C., Sainrat, P. and Triquet, B.: Predictable composition of memory accesses on many-core processors, *in Proc. of ECRTS* (2016).
[16] Saidi, S., Ernst, R., Uhrig, S., Theiling, H. and de Dinechin, B. D.: The shift to multicores in real-time and safety-critical systems, *in Proc. of CODES+ISSS*, pp. 220–229 (2015).
[17] Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Iyer, P., Singh, A., Jacob, T. et al.: An 80-tile 1.28 TFLOPS network-on-chip in 65nm CMOS, *in Proc. of ISSCC*, pp. 98–99 (2007).