Vol.2010-EMB-19 No.5
2010/12/6

組込み向け並列プログラミング言語の検討

阿部　睦[†]

近年，組込みシステムにおいてマルチコアプロセッサの導入が進み，マルチコアの能力を引き出すことが求められている．また，組込みシステムにおける開発規模の増大に伴い，開発効率向上も求められている．このような状況に対し，モジュールの構造と処理手順の定義によりプログラムを構成することを特徴とする並列プログラミング言語について開発を行った．開発したプログラミング言語は，単にマルチコアの活用のためだけでなく，導入の容易性や移植性，検証の容易性等，開発効率の向上にも対応できるものとなっている．本稿では開発した並列プログラミング言語の概要と試作について報告する．

A Study of Parallel Programming Language for Embedded Systems

Mutsumi Abe[†]

Recently, embedded systems with a multicore processor have increased; therefore it is required to exploit the capability of multicore processor. In addition, the development efficiency improvement is requested as the development scale increases in the embedded system. For this situation, I developed the parallel programming language which is characterized by programming with definition of the module's structure and procedures. This language developed with considering about portability and ease of deployment and ease of validation. This paper reports the language summary and its traial implementation.

1. はじめに

近年，組込みシステムにおいてもマルチコアプロセッサの導入が進んでおり，マルチコアプロセッサを活用するための研究開発が各所で行われている[1][2]．一方で，組込みシステムの開発規模が増大しており，一層の開発効率向上が求められており，マルチコアを対象としたソフトウェア開発支援についても提案されている[3].

このように，組込みシステムにおけるマルチコアをに向けたソフトウェア開発技術としては，単にマルチコアの性能を引き出すだけでなく，開発を効率化させることが要求される．具体的には以下のような項目が要件として挙げられる．

・導入の容易性
・移植性
・再利用性
・検証の容易性

導入の容易性を高めることにより，開発技術の導入によるコスト増を抑え，開発技術の持つコスト削減効果を高めることができる．導入の容易性を図る方法の一つとしては，開発者が少ない学習量で利用できることが挙げられる．

移植性を高めることにより，移植に伴うコストを抑制できる．移植性とは動作環境（プラットフォーム）が変わっても少ないコストで異なるプラットフォームで作成物を利用できることである．組込みシステムではいろいろなプラットフォームが想定されるため，それらへの対応も必要とされる．プラットフォームの違いとしてはプロセッサのコア数の違いだけでなく，ヘテロジニアス/ホモジニアスといったマルチコアプロセッサのアーキテクチャや専用ハードや OS の有無等が想定される．

再利用性は，一度作成したプログラムの一部または全部を他のプログラムの開発に利用できることである．これにより，派生システムの開発や同様の機能の取り込みが容易となる．

検証の容易性とは，プログラムが開発者の意図通り動作しているかを少ないコストで確認できることである．

本稿では，上記で述べた組込みシステムの開発に求められる要求に対応できる並列プログラミング言語の試作と評価について報告する．

本稿の構成は次の通りである．２章では検討した並列プログラミング言語の概要について述べる．３章では２章の内容に基づき試作した言語について述べる．４章では試作した言語の仮想マシンによる評価について述べる．５章では検討によって得られた各種課題について述べ，まとめる．

[†] 株式会社トヨタ IT 開発センター
Toyota InfoTechnology Center, Co., Ltd.

2. 検討した並列プログラミング言語概要

検討した並列プログラミング言語（以下，本言語と呼ぶ）について，構成要素と計算モデルについて述べ，本言語におけるプログラミングや並列処理への対応，１章で述べた開発効率における要件へのアプローチ等について述べる．

2.1 構成要素と計算モデル

本言語の構成要素はデータ構造を表すモジュールとモジュールの値を決定するメソッドである．計算モデルとしてはモジュールに対して，メソッドを適用することでモジュールの値を決定することとした．モジュール自体は型であり，メソッドはモジュールの値を決定することでモジュールに意味を与える手段である．

モジュールへのメソッドの適用時には他のモジュールを引数として参照することができ，メソッドと引数とメソッドを適用するモジュールの値が同じであれば，モジュールの値は一意に決まる．このように，モジュールは必要に応じて他のモジュールを参照して自身の値を決定するため，本言語ではデータ構造であるモジュール自体が他のモジュールに対するインタフェースとなっている．

なお，モジュールへのメソッド適用権限はモジュール自身かそのモジュールを内包する上位のモジュールからのみとし,モジュールへの操作における独立性を確保した．

このような構成要素と計算モデルとしたのは，データ（モジュールの値）を中心に考えることで,並列処理の記述の容易性等の各種利点が得られると考えたためである．

また，モジュールは計算処理だけでなく，モジュールの参照を入力，モジュールの値を決定することを出力という意味で捉え，入力や出力を担うモジュールを用意することで入出力処理にも対応する．

2.2 プログラミング

本言語でのプログラミングは，モジュールの定義と定義されたモジュールに対するメソッドの定義と定義したモジュールを実体化し，実体化したモジュールにメソッドを適用し値を決定することで処理を進める部分から構成される．

モジュールの定義は定義済みのモジュールを組み合わせて定義を追加することにより行われる．メソッドも同様に定義済みのメソッドを組み合わせて定義することにより追加する．メソッドの定義は定義済みモジュールに対して行われる．実体化したモジュールはインスタンスと呼び，モジュールの実体化はインスタンスの宣言を定義済みのモジュールを用いて行う．

本言語の試作によるプログラミングの詳細については３章で述べるが，他のプログラミング言語との違いとして，四則演算等の基本的な演算だけでなく，制御構造やインスタンスへの値の設定（代入）もメソッドとして定義することが挙げられる．

2.3 本言語における並列処理

並列処理は，メソッドの適用について逐次/並列を陽に指定することで記述する．

並列にメソッドの適用を行う際に下記のルールとすることで本言語における通常のプログラミングでは排他制御の必要性を排除している．

- 書込みの無いインスタンスは読込み可能
- 同一のインスタンスへの複数の書込みは禁止
- 書込みのあるインスタンスに対しては利用者による指定時を除いて読込み禁止

これにより，インスタンスの処理は基本的にはブロックされることなく実行できるため，並列処理可能なインスタンス数がプロセッサコア数より大きい場合はプロセッサ（コア）数に応じたスケーラビリティが期待できる．

並列処理の粒度に関しては，本言語でのモジュールの最小単位はビットレベルを想定しており,モジュールは定義により必要に応じて大きくしていくことができるので,並列処理の粒度に関してもスケーラビリティがあると考える．

本言語の処理はモジュール（インスタンス）の値を決定することであるので，モジュールの実際の処理においては空いているプロセッサ（コア）に対して未処理のインスタンスを割り当てるだけでよい．そのため，スレッドを用いた並列処理と比較すると，スレッド管理のオーバーヘッドの低減が期待できる．

2.4 本言語の実行と仮想マシン

本言語で書かれたプログラムはコンパイラにより中間コードに変換され，変換された中間コードを仮想マシンが解釈し実行する．ここでの仮想マシンは広義の仮想マシンである[4]．中間コードは，定義されたモジュールやメソッドを次に述べる基本的なモジュールとそのメソッドにまで展開し，コード化した形で表現される．仮想マシンでの中間コードの処理は，プログラム中のインスタンスをメモリに展開し，メソッドの処理内容を動作するプロセッサ（コア）へ割り当てることで行う．

仮想マシンの構築は，数値や文字列等，動作上必要とされるデータまたはデータ構造を基本的なモジュールとし，それらに対する処理方法であるメソッドをターゲットとなるプラットフォームの開発用言語（例えば，C やアセンブラ）で記述することで行われる．コンソールへの出力等，基本的な入出力動作も基本的なモジュールとして実現する．

なお，本言語のコンパイラでは，一般的なコンパイラでは通常行われる最適化[5][6]は現時点では適用しない予定である．これは，プラットフォームの構成により適切な最適化が異なるため，本言語のコンパイラで対応するよりも，仮想マシンを構築する段階で基本的なモジュールの実装で対応したほうがよいと考えたためである．

2.5 組込みに向けた特徴

ここでは，本言語の組込みに向けた特徴について述べる．

（1）プラットフォームの違いの吸収

プラットフォームの違いの吸収は，仮想マシンの構築に必要な基本的なモジュールのデータ構造とメソッドをプラットフォーム毎に用意し，仮想マシンを移植すること

で実現する．

専用プロセッサや GPU 等を利用したい場合は，各種プロセッサの処理結果で得られるデータ構造をモジュールで定義しておき，メインのプロセッサと各種プロセッサとのやりとり（引数や結果の受渡し，処理指示等）を処理結果のモジュールに対するメソッドとしてプラットフォームの開発用言語で記述することで可能となる．

また，仮想マシン構築のための基本的なモジュール以外にも高速化したいデータ処理についても同様にプラットフォームの開発用言語で記述することで高速化が狙える．

（２）リソース制約への対応

仮想マシンはシステムで必要とされる情報を表現するモジュールがあれば良いので，コンパクトなものから機能豊富なものまで用途に応じて構築することができる．

これにより，仮想マシンの移植がプラットフォームの規模に応じたサイズで実現できると考え，リソース制約の厳しい組込みでも仮想マシンによる対応が可能と考える．

また，仮想マシンでの動作ではなく，プラットフォームの開発用言語のコードでコンパイルして実行したい場合は，コンパイラで中間コードを出力する替わりに，仮想マシン構築用に定義されたモジュールとメソッドについて開発用言語に変換する機能を用意し，ユーザプログラムと共に開発用言語に変換することで対応が可能と考える．

（３）ハードウェア/ソフトウェア協調設計

現在の組込みシステムの開発では，ハードウェア/ソフトウェア協調設計の流れもあり[7]，通常のプログラミング言語以外にもハードウェア記述言語への展開も考えられる．C 言語ベースのハードウェア設計技術も提供されているが，並列処理が自然に記述でき，データ粒度を細かく設定可能な本言語はハードウェア記述言語へ比較的素直に変換できると考えている．

2.6 開発効率における要件へのアプローチ

ここでは，１章で述べた開発効率における要件への本言語のアプローチについて述べる．

（１）導入の容易性

本言語における基本的な構成要素であるモジュール自体は単なるデータ構造であり，構成としては C 言語の構造体と変わらない．また，メソッドの定義も対象とするモジュールの値を決めるための処理を記述していくことなので，結果を戻り値として返す C 言語の関数定義と変わらない．

このように，本言語の構成要素は C 言語を理解している利用者であれば，容易に理解できると思われ，プログラミングにおいても比較的抵抗無くプログラムを書くことが可能と思われる．よって，習得に要するコストは低いと考えられる．

また，構文自体のわかりやすさについては，次章で本言語の実装例について述べるが，メソッド定義が柔軟であるため，組込みシステムでよく利用される C 言語の構文に近い形でメソッドを定義することも可能である．

これらにより，C 言語に慣れた利用者であれば，導入は容易と考える．

（２）移植性

本言語は仮想マシンでの動作を主として考えており，本言語で作成されたプログラムの移植性は高いと考える．

（３）再利用性

2.1 節で述べたように，モジュール自体がインタフェースとなるため，モジュールをソフトウェア部品として考えることが可能である．

また，機能を呼ぶ形式でなく，参照によって他のモジュールと関係を持つため，モジュールの組合せにおける接続性が高いと考える．

（４）検証の容易性

モジュールの処理の確認については，対象とするモジュールのインスタンスとメソッドと引数（参照するインスタンス）の値が同じであれば，インスタンスの値は一意に決まるため，テストパターンの作成が容易であり，粒度の小さいモジュールでは，参照するモジュールの値と結果の全パターンをチェックすることも可能である．

また，本言語では処理の結果が直接モジュールの値に反映されるので，値を決定する処理手順であるメソッドがブラックボックスとして提供されたとしても，結果のモジュールと参照されるモジュールの値の対応が正しければよいので，受け入れ検査が容易となると考えられる．

3. 試作した並列プログラミング言語

２章の内容に基づいて試作した並列プログラミング言語について以下述べる．

3.1 構文の文法

構文の文法については，付録として EBNF による文法を掲載している．次節以降の説明では，言語の構成要素と構文を併せて説明しているため，必要に応じて参照されたい．

3.2 試作した本言語におけるプログラムの構成

試作した本言語におけるプログラムは，モジュール定義部，メソッド定義部，実行部の 3 つで構成する．以下では，これらの構成部分と構成要素，逐次/並列の表現方法等の特徴的な点について述べる．

3.2.1 モジュール定義部

新しいモジュールは既に定義済みのモジュールを用いて定義する．本言語では，処理系として与えられる定義済みモジュールを基本モジュールと呼ぶ．

本言語で必須とした基本モジュールは Bit モジュールで 0,1 の二つの値のどちらかを取る．

他の基本モジュールとして，プログラム構造を表すモジュールとして Block モジュ

ールがある．Block モジュールは，波括弧{}で囲まれたものとして表現する．Block についての詳細は後述する．

モジュールの定義は，予約語「def」に続く文字列がモジュール名となり，波括弧で括られた内部にそのモジュールが内包するモジュールを宣言し定義することが出来る．

32 ビット整数のモジュールの定義例を以下に示す．

```
def U32 { Bit bit[32]; };
```

ここで，bit[32] は Bit モジュールの配列を示し，Bit モジュール 32 個分の領域が確保される．配列はインスタンス名と角括弧([])で要素を表す添字を括る形式で表現する．

モジュールを定義する場合，その構成要素となるモジュールを内包モジュールと呼ぶ．内包モジュールを参照する場合，「.（ピリオド）」を用いる．複数のモジュールから構成されるモジュールの定義と内包モジュールの参照例を以下に示す．なお，”//” は行コメントである．コメントとしては C 言語と同様に”/*”～”*/”も利用可能である．

```
def Point { //  モジュール定義宣言とモジュール名
        S32 x; // 32 ビット符号付整数の内包モジュール
        S32 y; // 32 ビット符号付整数の内包モジュール
}; //  モジュール定義完了
S32 dx; // 32 ビット符号付整数のインスタンス宣言
Point delta; // Point モジュールのインスタンス宣言
dx = delta.x; // delta の内包モジュール値のセット
```

3.2.2 メソッド定義部

メソッドの追加定義もモジュールの定義と同様に，定義済みのモジュールとメソッドを用いて定義する．メソッドの定義は対象とするモジュールが既に定義されていればどの時点で定義しても構わない．

仮想マシンにおける基本的なモジュールに対するメソッドは定義済みメソッドとして，プラットフォームの開発用言語で記述される．このようにプラットフォームの開発用言語で記述されたメソッドをプリミティブメソッドと呼ぶ．

定義されたメソッドはコンパイル時にコンパイラによってプリミティブメソッドまで展開され，中間コードにはプリミティブメソッドの識別情報が格納され，仮想マシンでの実行で利用する．

プリミティブメソッドとして，今回の試作では基本モジュールである Bit モジュールには論理和，論理積，否定，排他的論理和の論理演算と値の設定（代入）がプリミティブメソッドを用意した．

また，プリミティブメソッドはユーザによる定義が可能なので，高速化等を想定し，指定されたデータ構造に対してユーザ定義のプリミティブメソッドを作成することが可能である．また，メインのプロセッサと専用のプロセッサとのやりとり等をプリミティブメソッドで記述することにより，プラットフォームの特殊性を隠蔽することも可能である．

今回の試作では，ユーザ定義によるプリミティブメソッドとして，C 言語における基本的なデータ型である，char, short int, int, long について符号有無分も含めてプリミティブメソッドとして定義した．

メソッドは 1 つのモジュールに対して複数定義することが可能である．定義対象のモジュールが異なっていれば，同一名称のメソッドも定義可能である．

メソッド定義内の評価式によって，対象となるモジュールのインスタンスの値を変更することになるが，その際，対象となるモジュールのインスタンスの値を表す予約語として，「self」がある．

メソッドの定義例として，上記で定義した Point モジュールに対する初期化メソッドを定義してみる．

```
method Point(‘init’){
        self.x = 0;
        self.y = 0;
}:
```

なお，本言語では，’=’や四則演算，論理演算等の演算記号もメソッドの追加定義やプリミティブメソッドの定義に使用可能となっており，プラットフォームに合わせた定義が可能となっている．

3.2.3 実行部

本言語におけるプログラムの実行はモジュールのインスタンスにメソッドを適用し，インスタンスの値を確定することである．本言語では，インスタンスへのメソッド適用をインスタンスの評価，または単に評価と呼ぶ．

3.2.4 インスタンス

インスタンスとは，定義済みのモジュールを実体化したものであり，宣言に使用されたモジュールに応じたサイズの記憶領域が確保される．

インスタンスへのメソッドの適用は，インスタンス宣言後にそのインスタンスに続けてメソッドを記述することで行う．なお，メソッドが適用されるインスタンスをターゲットインスタンスと呼ぶ．

3.2.5 逐次処理と並列処理

インスタンスの評価を逐次に行う場合は「;（セミコロン）」，並列に評価する場合は「,（カンマ）」で区切ることにより評価の順序指定を行う．

「;（セミコロン）」の後に配置された評価式は，前の評価式の評価が終了するまで評価が始まらない．

一方，「,（カンマ）」の後に配置された評価式は，前の評価式の評価終了に関わらず，並列に評価される．但し，逐次間の適用順序は保証されるが，並列の評価では評価の順序は不定である．

並列に評価するインスタンスが複数ある場合，並列の評価とした全てのインスタンスの評価が終了した後に，後続の逐次評価が開始する．下記の例の場合，a，b，c の評価が全て終了した後に d の評価が開始する．

```
S32 a,b,c,d;
a = 1,
b =2,
c = 3;
d = a + b + c;
```

上記では，a = 1 から c = 1 までが並列処理可能となり，a，b，c の値が確定した後，d の評価が開始され値が決定する．

3.2.6 ブロック

ブロックはプログラム構造を表すもので，波括弧で囲まれた部分を言う．ブロックもまたモジュールである．

ブロックもまたモジュールのため，インスタンスとして宣言することができる．ブロックのインスタンスに対しては，ブロックを値として設定することができる．

ブロックには下記のメソッドが使用可能である．

```
eval // インスタンスのブロックを評価する
set // インスタンスにブロックデータをセットする
```

ブロックの記述例を以下に示す．

```
Block prg;
prg set {
        S32 a,b,c;
        a = 1; b = 2; c = a + b;
};
prg eval;
```

ブロックをモジュールとした理由としては，プログラム自体をデータ化するためである．ブロックをモジュールとすることで，制御構造も本言語のプログラムとして記述することが可能である．

3.2.7 ブロックにおける逐次と並列

逐次／並列実行については，ブロックについても逐次実行を「;（セミコロン）」，並列実行を「,（カンマ）」で区切ることにより指定可能である．

例えば下記のように記述した場合，a，b を含む波括弧の評価が全て終了するまで，c，d を含む波括弧の評価は開始しない．

```
S32 a,b,c,d;
{
        { a = 1; b = 2};
        { c = 3; d = 4; };
};
```

また，下記のように記述した場合，a，b を含む波括弧の評価と，c，d を含む波括弧の評価を並列に実行する．

```
{
        { a = 1; b = 2},
        { c = 3; d = 4; };
};
```

3.2.8 値の参照

並列評価時の非決定性を排除するために，「,（カンマ）」で区切られた並列評価の記述において，あるターゲットインスタンスを並列評価の他の評価式のターゲットや引数にすることはできずコンパイルエラーとなる．

しかしながら，評価中のインスタンスの値を知りたい場合も想定される．その場合，「@（アットマーク）」を評価中のインスタンス名の前に付けることにより，引数として使用できる．

この「@」を付加した参照を非同期参照と呼ぶ．非同期参照は，インスタンスの値決定に対して非決定性を持ち込むため，使用は最低限に抑えるべきである．

非同期参照におけるターゲットインスタンスの内部インスタンスへの参照は丸括弧によって非同期範囲を明示する．その際，参照対象の内部インスタンスにメソッドが適用されている場合は適用が終了するまで参照は待たされる．参照の具体例は以下のようになる．

```
def Point { U32 x,y;};
def Rectangle { Point leftup,rightdown;};
method Rectangle ( 'move'){/*  何らかの移動処理  */ };   // Rectangle のメソッド例
Rectangle a; U32 b ; Point c,d;
a move,
b = @(a.leftup.x),      // 最後まで非同期参照の場合
c = @(a).leftup,        // leftup に対してメソッドが適用されている場合は参照は待たされる．
d = @a.leftup;          // @(a).leftup と同じ
```

ここで，b は直ちに値をセットされるが，c は leftup が確定するまで待たされ，d は c と同様である．

これは，あるモジュールが巨大であり，並列動作するモジュールがそのモジュールの内部のモジュールを参照する場合は，内部のモジュールの値が確定した段階での読み込みを許すことで，参照するモジュールの処理を待たずに並列に動作させたいためである．また，あるモジュールが常に動作中で，その内部の情報が知りたい場合等に

も利用できる．なお，非同期参照については今後見直す可能性がある．

3.2.9 プリプロセッサ

定義済みのモジュールの読み込みや値の名前付けとして，#include, #define を用意した．コンパイル時のコメントの除去もプリプロセッサで行う．
プリプロセス時には後述する不定長メソッドの展開も行う．

3.2.10 不定長メソッド

メソッド定義の引数の最後に’$’マークを配置することにより，不定長メソッドとして定義できる．使い方としては，四則演算の優先度の定義や本言語では制御構造の定義での利用を想定している．

ここでは四則演算の優先度を例に取って説明する．

```
S32 a,b,c,d,e;
a = b + c * d - e;                    （１）式
```

このとき（１）式の演算の優先度としては括弧で括ると，次式のようにしたい．

```
a = b + (c * d) - e;
```

このような演算の優先度は不定長メソッドの定義を用いて以下の様に定義できる．

```
method S32 ('=',S32 arg1,'+',S32 arg2,'*',S32 arg3,'-', $){ // $ は（１）では e に相当
        S32 x, y;
        x = arg1,
        y = arg2;
        y = y * arg3; // 乗算を先に実行
        x = x + y; // 加算を実行
        self = x - $; //   $ は e として展開
};
```

引数の最後の‘$ ’がメソッド定義時の残りの引数パターンをあらわす記号で，プリプロセス時に’$’に相当するメソッドのパターンはそのまま展開される．なお，この定義においては'-'以降も不定長の可能性があるので，減算の定義においても本定義と同様の定義が必要である．

3.2.11 制御構造

本言語では，制御構造もモジュール定義とブロックを用いて定義する．これは，本言語ではコンパイラによる最適化を適用しない予定であり，制御構造自体も本言語のプログラム上で定義できる可能性を残すためである．例えば，あるインスタンスに対し，条件の真偽によって格納する値が決まるような条件文では，条件判断と真偽それぞれのブロックを並列に実行するように投機的な実行を本言語のプログラム上で実現することも可能である．

また，制御構造自体も定義により構築されるため，プリミティブメソッドとして定義することも可能であり,プラットフォームに合わせた制御構造の定義も可能となる．

制御構造をモジュール定義とブロックで実現するため，まずは真偽値を表すモジュールとして Boolean を定義する．Boolean は Bit モジュールで定義され，真偽値として true,false の二つの値を持つ．true,false の実体は,1 と 0 の#define である．定義を以下に示す．

```
def Boolean { Bit boolean; };
#define     true        0
#define     false       1
```

（１）条件文の例

条件文である if~else 文は下記ように定義できる．

```
method Boolean (‘then’, Block truebody, ‘else’, Block falsebody ){
        Block body[Boolean]; // ブロックの配列
        body[true] set truebody; // 真のときの実行ブロック
        body[false] set falsebody;
        // 偽のときの実行ブロック
        body[self] eval;
        // 該当するブロックの実行
};
#define     if          Boolean
```

この定義を使ったプログラム例は以下の通り．

```
S32 a,b;
a = 10,b = 3;
if (a > b) then { a -= b;} else { a += b;};
```

先頭の if は Boolean となり，(a>b)は Boolean の一時変数として a>b の結果が格納され，上記のメソッド定義が実行される．

（２）繰り返し

何らかの繰り返し処理を行いたい場合は，実行中のメソッドを最初から実行しなおす ‘redo’ を使用可能とした．但し，繰り返しの実現方法については今後見直す可能性がある．

（３）その他の制御構造

C 言語のおける switch～case 文のような不定長の制御構造は条件文の定義のやり方と不定長メソッドの定義を用いることで定義可能である．

4. 評価

ここでは，試作したコンパイラと仮想マシンについて概要を述べ，仮想マシンでの性能評価について述べる．

4.1 試作したコンパイラ

今回試作したコンパイラは仮想マシンで実行する中間コードを出力する．なお，今回は動作検証レベルの試作であり，3 章で説明した言語仕様全てはカバーしていない．

4.2 試作した仮想マシン

仮想マシンもコンパイラと同様，動作検証の位置づけで試作を行っており，コンパイルされた中間コードのロードや実行等の最低限の機能のみを持つ．仮想マシンの実装は C 言語で行った．

仮想マシンの動作としては，内部でスレッドを動作させ，中間コードで並列部分がある場合に並列処理可能なインスタンスの評価にスレッドを割り当てることを行っている．並列処理するインスタンス数がスレッド数を上回っている場合は，処理が終了したスレッドから処理待ちのインスタンスの処理を行う．これにより，無駄なくスレッドを動作させることができる．なお，内部で動作するスレッド数は設定により変更することが可能である．

仮想マシンの内部スレッド数が同時実行可能なスレッド数以内であれば，基本的にはスレッドの切替が起こらず，並列処理可能なインタスタンス数に応じてスレッドが動作するため，効率的な動作が期待できる．

4.3 測定による評価

測定による評価は同じ題材について，C 言語および本言語でプログラムを作成し，処理速度の比較とスレッド数増加に対する速度改善を確認することで行った．なお，仮想マシンにおいて測定に必要な時間出力機能はプリミティブメソッドを用いて実現しており，仮想マシン内で特に測定機能は持たせていない．

測定のための評価環境として使用したマシンの仕様は次の通りである．

プロセッサ：Intel Core i7-920
メモリ：6GB
OS：OpenSolaris

今回の仕様したマシンのプロセッサは 4 コア搭載し，ハイパースレッディングテクノロジにより，8 スレッド同時動作を可能としている．

プログラムの題材としては，並列性の高い画像処理を想定し，画像のエッジ検出を対象とした．エッジ検出のプログラムを C 言語と本言語で記述し，スレッド数および画像サイズを変化させて測定を行った．

測定結果で示すスレッド数は，仮想マシン内で使用しているスレッド数としている．

画像サイズは横 160 縦 120 から横 160 縦 120 刻みで横 1600 縦 1200 の UXGA サイズまでとした．

結果としては，仮想マシン内のスレッド数に応じて処理能力が上がっており，スケーラビリティは実現されているといえる．しかしながら，処理速度については本言語が C 言語に比べて 3 桁程度遅い結果となった．

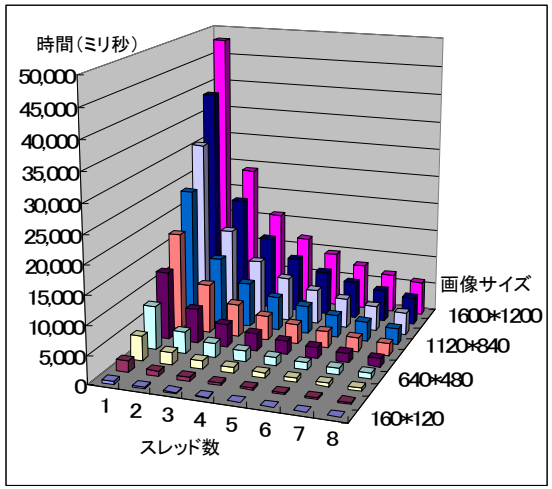

図 1　試作した言語での測定結果
Fig.1 Performance result in trial implementation language

5. まとめ

本稿では，マルチコアに対応しつつ組込みシステム開発における要件に対応可能な並列プログラミング言語の概要と動作検証のための試作による評価について報告した．

マルチコアへの対応は，モジュールに基く計算モデルにより，並列動作可能な処理を明確化することで対応した．

開発効率の要件への対応については，導入の容易性，移植性，再利用性，検証の容易性について，それぞれに対応するアプローチについて述べた．

試作を通じて明らかになった課題としては，制御構造対応のための繰返し命令等，言語構造の一貫性が保たれていない可能性等，構文定義の検討が不足している点が挙げられる．

また，試作した仮想マシンにおいては，測定による評価で見たとおり，スケーラビリティは確保できているが，現時点では全く性能が出ていないため，原因を特定後，コンパイラや仮想マシンの見なおしが必要と思われる．

今後の取組みとしては，明らかになった課題に対し，構文定義の見直しやコンパイラおよび仮想マシンの再構築を実施する予定である．

参考文献

1) 高山征大，境　隆二，加藤宣弘，島田智文：並列プログラミングモデル Molatomium，情報処理学会論文誌：プログラミング, Vol.3, No.1, pp.54-62(2010).
2) Keiji Kimura, Masayoshi Mase, Hiroki Mikami, Takamichi Miyamoto, Jun Shirako and Hironori Kasahara, :OSCAR API for Real-time Low-Power Multicores and Its Performance on Multicores and SMP Servers, Proc. of The 22nd International Workshop on Languages and Compilers for Parallel Computing , (2009).
3) 高橋清隆，柴山悦哉：組込みシステム向けマルチコア・プロセッサのためのソフトウェア開発支援，情報処理学会論文誌：プログラミング,Vol.48,　No.SIG 4(PRO 32), pp.27-47(2007).
4) 高野了成，石口栄一，早川栄一，高橋延匡：仮想マシンを用いた OS 構成法とその通信機構，情報処理学会研究報告，Vol.2002,No.60,pp.127-132(2002).
5) 中田育男，渡邊　坦：２１世紀のコンパイラ道しるべ‥COINS をベースにして「概要」，情報処理，Vol.47,No.4,pp.425-435(2006).
6) University of Illinois:The LLVM Compiler Infrastructure. http://llvm.org/
7) 本田晋也，冨山宏之，高田広章：システムレベル設計環境：SystemBuilder，電子情報通信学会, D-I,Vol.J88-D-I,No.2,pp.163-174(2005).

付録

付録 A.1 EBNF による本言語の文法

program = program_elements | program, program_elements ;
program_elements = declaration | expression_list ;
declaration = module_declaration, separator | method_declaration, separator | instance_declaration, separator ;
separator = [white_space], ';', [white_space] ;
(* module declaration *)
module_declaration = DEFMODULE, white_space, module_name, left_brace, instance_declaration_list, separator, right_brace ;
left_brace = [white_space], '{', [white_space] ;
right_brace = [white_space], '}', [white_space] ;
DEFMODULE = 'def' ;
instance_declaration_list = instance_declaration | instance_declaration_list, separator, instance_declaration ;
module_name = BASIC_MODULE | name_exp | array_def_exp ;
BASIC_MODULE = 'Bit' | 'Block' ;
name_exp = name_characters - digit | name_exp , name_characters ;
array_def_exp = name_exp, '[', ']' ;
(* method declaration *)
method_declaration = METHOD, white_space, module_name, left_paren, method_arg_list, [comma_separator , '$',] right_paren, method_body_exp;
METHOD = 'method' ;
left_paren = [white_space], '(', [white_space] ;
right_paren = [white_space], ')', [white_space] ;
method_arg_list = method_arg | method_arg_list, comma_separator , method_arg ;
method_arg = """, method_string - comment_symbols,""" | module_name, white_space, instance_name;
method_string = method_characters | method_string , method_characters ;
method_characters = name_exp | symbols ;
method_body_exp = left_brace, right_brace (* body is nil *) |left_brace, method_body_element_list, right_brace ;
method_body_element_list = method_body_elements | method_body_element_list, method_body_elements ;
method_body_elements = instance_declaration, separator | METHOD_REDO, separator | expression_list | left_brace, method_body_element_list, right_brace, separator (* nesting *) ;
METHOD_REDO = 'redo' ;
(* instance declaration *)
instance_declaration = module_name, white_space, instance_name_list ;
instance_name_list = name_exp | instance_name_list, comma_separator , name_exp | array_declaration | instance_name_list, comma_separator , array_declaration ;
array_declaration = name_exp, '[', [white_space], number, [white_space], ']' | name_exp, '[', [white_space], module_name, [white_space], ']' ;
instance_name = name_exp | name_exp, '.', name_exp ;
comma_separator = [white_space], ',', [white_space] ;
array_exp = name_exp, '[', [white_space], array_index, [white_space], ']' ;
array_index = number | module_name | instance_name | array_exp ;
(* expression_list *)
expression_list = sequential_exp_list, separator | expression_list, sequentail_exp_list, separator | parallel_exp_list, separator | expression_list, parallel_exp_list, separator ;
sequential_exp_list = instance_exp | sequential_exp_list, separator, instance_exp ;
parallel_exp_list = instance_exp, comma_separator , instance_exp | parallel_exp_list, comma_separator , instance_exp ;
instance_exp = instance , method_exp | array_exp, method_exp | module_name, left_paren, method_exp ,right_paren, method_exp | block_exp | block_exp, method_exp ;

block_exp = left_brace, right_brace | left_brace, expression_list, right_brace | left_brace, block_exp, separator, right_brace ;
instance = 'self' | instance_name | array_exp | left_paren, instance_exp, right_paren ;
method_exp = method_element_list, [whitespace,'$'] ;
method_element_list = method_elements | method_element_list, method_elements ;
method_elements = white_space, method_data_elements | symbol_only_method_string | symbol_first_last_method_string | symbol_first_method_string, white_space | white_space, symbol_last_method_string ;
method_data_elements = instance_or_array | async_access_exp | array_data | number | left_paren, module_name, white_space, number, right_paren | block_exp ;
symbol_only_method_string = symbols | symbol_only_method_string, symbols ;
symbol_first_last_method_string = symbols, method_string, symbols ;
symbol_first_method_string = symbols, method_string, name_exp ;
symbol_last_method_string = name_exp, method_string, symbols ;
instance_or_array = instance_name | array_exp | instance_or_array, '.', instance_or_array ;
async_access_exp = '@',instance_or_array | '(', [white_space], '@', instance_or_array, [white_space], ')', '.', instance_or_array ;
array_data = left_brace, array_data_element_list, right_brace | data_string ;
array_data_element_list = array_data_elements | array_data_element_list, comma_separator, array_data_elements ;
array_data_elements = number | char_const ;
char_const = "'", data_characters, "'" ;
data_string = '"', data_str_char_list, '"' ;
data_str_char_list = data_str_char | data_str_char_list, data_str_char ;
data_str_char = data_characters - '"' ;
(* pre_processor *)
PRE_PRO_SOURCE = PRE_PRO_SOURCE_ELEMENT | PRE_PRO_SOURCE, PRE_PRO_SOURCE_ELEMENT ;
PRE_PRO_SOURCE_ELEMENT = program | PRE_PRO_INCLUDE | PRE_PRO_DEFINE ;
PRE_PRO_INCLUDE = '#include', [pre_pro_separetor], '"', include_file_name, '"', pre_pro_end_of_line ;
include_file_name = ? File Name ?;
PRE_PRO_DEFINE = '#define', pre_pro_separator, name_exp, pre_pro_separator, pre_pro_value, pre_pro_end_of_line ;
pre_pro_value = area_commentstring - line_commnet_start - end_of_line;
pre_pro_separator = pre_pro_separete_char | pre_pro_separete_char, pre_proseparator ;
pre_pro_separate_char = white_spase_character - end_of_line ;
pre_pro_end_of_line = line_comment | line_area_comment, [pre_pro_separate_char], end_of_line | [pre_pro_separate_char], end_of_line ;
(* comment *)
comment = line_comment | area_comment ;
line_comment = line_comment_start, [all_character_string - end_of_line], end_of_line ;
line_comment_start = '//' ;
area_comment = comment_start, area_comment_string, comment_end ;
area_commnet_string = all_character_string - comment_start - comment_end ;
comment_start = '/*' ; comment_end = '*/' ;
comment_symbols = line_comment_start | comment_start | comment_end ;
line_area_comment = comment_start, area_comment_string, comment_end ;
line_area_commnet_string = all_character_string - comment_start - end_of_line ;
all_character_string = data_characters | all_character_string, data_characters ;
(* numbers and characters *)
number = ["-"],decimal | hexadecimal | binary ;
decimal = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | decimal,digit ;
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
hexadecimal = '0x', hex ;
hex = hex_digit | hex, hex_digit ;
hex_digit = digit | 'a'| 'b'| 'c'| 'd'| 'e'| 'f'| 'A'| 'B'| 'C'| 'D'| 'E'| 'F' ;
binary = '0B', bin | '0b',bin ;
bin = '0' | '1' | bin, '0' | bin, '1' ;
white_space = comment | white_space_characters | white_space, white_space_characters ;
white_space_characters = ' ' | #x9 (* tab *) | end_of_line ;
end_of_line = #xd (* cr *) | #xa (* lf *) | #xd,#xa (* cr,lf *) ;
data_characters = symbols | reserved_symbols | name_characters | white_space_characters ;
symbols = '=' | '+' | '-' | '*' | '/' | '&' | '|' | '!' | '^' | '%' | '<' | '>' | '~' | ':' | '?' ;
reserved_symbols = ';' | ',' | '.' | '{' | '}' | '[' | ']' | '(' | ')' | '@' | "'" | '"' | '$' | '#' | '¥' ;
upper_letters = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' ;
lower_letters = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' ;
name_characters = digit | upper_letters | lower_letters | other_symbols ;
other_symbols = '_' ;