情報処理学会論文誌　プログラミング　Vol. 3　No. 1　54–62 (Mar. 2010)

並列プログラミングモデル Molatomium

高山征大[†1]　境　隆二[†1]
加藤宣弘[†1]　島田智文[†1]

マルチコア時代に向けた，実行性能の高い並列プログラムを容易に書くための並列プログラミングモデル *Molatomium* と，その処理系および開発支援ツールを提案する．プログラミング言語としては，並列性を記述する C 言語風の *Mol* という言語と，実行性能を追求する直列コード *Atom* を記述する C/C++ を併用する．Mol は，データ並列およびタスク並列を扱う．配列によってデータの依存関係を遅延代入式で表現するとともにメモ化を行い，データ並列を実現する．依存関係のない関数は自動的に並列に実行され，タスク並列を実現する．Mol コードはバイトコードに，Atom コードはネイティブコードにコンパイルされ，1 つのプログラムをなす．Atom を組み上げて Mol を構成するというアナロジである．処理系は仮想マシンとして実装し，プラットフォーム間における差異を吸収する．Mol で記述した依存関係に従って Atom を動的にスケジューリングし，並列に実行する．x86 で動くいくつかの POSIX OS（Linux，Mac，Cygwin）および Cell Broadband Engine™（以下 Cell/B.E.），SpursEngine™ に処理系を実装し，それぞれでコア数に応じたスケーラビリティを確認した．また，並列プログラミングを支援するためのグラフィカルなエディタ，デバッガ，三次元可視化環境を紹介する．

Molatomium: A Parallel Programming Model

Motohiro Takayama,[†1] Ryuji Sakai,[†1]
Nobuhiro Kato[†1] and Tomofumi Shimada[†1]

We propose a parallel programming model *Molatomium*, its runtime system, and its development tools to create effective parallel program easily for multi-core era. We use both a C-like language named *Mol* that describes concurrency and C/C++ that describes high performance serial code *Atom*. Mol is responsible for both data and task parallelism. It supports data parallelism by using arrays to represent data dependency flow and to memoize. It also supports task parallelism by parallel execution of functions that has no dependencies. Mol code is compiled into byte code and Atom code is complied into native one, that results in composing a parallel program, as a molecule is composed by atoms. The runtime system is implemented as a virtual machine to achieve cross platform compatibility. It schedules atom executions as described in Mol code, and run them concurrently. We implemented the runtime system on some x86 POSIX platforms (Linux, Mac, Cygwin) and Cell/B.E., SpursEngine. Each implementation scaled as the core count increased. Furthermore, we introduce a graphical editor, debugger, and 3-D visualization to support efficient parallel programming.

1. はじめに

マルチコアプラットフォームで動作する効率的な並列プログラムを開発するのは困難である．この困難さは，今後拡大するヘテロジニアスマルチコア時代になると，さらに増大することが必至である．我々が提案する並列プログラミングモデル *Molatomium* は，この問題に対する実用的なアプローチである．すなわち，C プログラマに親しみやすい構文を持ち，高速なコードが書け，移植性が高く，スケーラブルであることを目標とする．

Molatomium は **図 1** の要素からなる．*Mol* という C 言語に似た構文を持つ言語で並列性を表現し，*Atom* という C/C++ で書かれたコードで直列性能を追求する．Mol と Atom という 2 つのプログラム要素を組み合わせることで，並列プログラミングの容易さと実効性能とをバランス良く実現する．Runtime（処理系）は各プラットフォームごとに用意され，プラットフォーム間の差異を吸収するとともにスケーラビリティを提供する．x86 プラットフォームおよび Cell/B.E.，SpursEngine に処理系を実装しており，共通の Mol コードを用いたクロスプラットフォームな開発を行えている．

並列プログラムをテキストだけで考え，デバッグするのは困難である．そのため，プログラミングモデルに加え，並列アルゴリズムの設計を支援するためのエディタ，並列プログラムの問題を追跡するためのデバッガ，実行を追うための可視化といったグラフィカルな開発ツールを用意した．

次章以降は，以下のような構成をとる：まず 2 章でプログラミングモデルの設計を述べ，3 章で処理系の設計および実装に触れる．4 章では開発ツールを紹介し，5 章で評価結果を述べる．6 章で関連研究を紹介し，7 章でまとめる．

†1 株式会社東芝ディジタルメディアネットワーク社コアテクノロジーセンター
Core Technology Center of TOSHIBA CORPORATION Digital Media Network Company

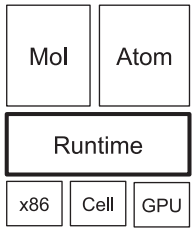


図 1　構成図
Fig. 1　Building block.

2. プログラミングモデル

我々のプログラミングモデルについての要求には以下のものがある：

実行性能の高いコードが書けること． 組み込み機器の開発に従事している我々のミッションは，高度に最適化された実行効率の高いコードを提供することである．

学習しやすいこと． ターゲットとなる開発者は日々忙しく，すでに馴染んでいる言語（C）と乖離しすぎていると使ってもらえない．

スケーラビリティ． 開発者は，ターゲットとなるプロセッサ，コア数が変わるたびにソフトウェアを書き直すことに辟易している．

こうした要件を満たすように設計したプログラミングモデルが Molatomium である．Molatomium は *Mol* と *Atom* という 2 つの言語要素からなる．Mol + Atom で Molatomium である．

Mol はデータ並列，タスク並列の両方に責任を負う．配列を用いてデータの依存関係を表現し，データ並列を実現するとともに，メモ化による計算のキャッシュを行う．また，依存関係のない関数を並列に実行することによって，タスク並列の能力を提供する．

Atom は，高い性能を達成するためのプラットフォームネイティブな直列コードであり，C/C++ で記述する関数である．開発者は，SIMD 命令などを駆使してプラットフォームごとに最適化したコードを書く．

Mol のコードはプラットフォームに依存しないバイトコードにコンパイルされ，Atom のコードはネイティブコードに落ちる．ちょうど分子（Molecule）が原子（Atom）から構成されるように，Atom と Mol が組み合わさることで並列プログラムができあがる．

図 2 は，Molatomium を用いて Fibonacci 数を計算する Mol，Atom コードである．以下，このコードを例にして Mol の言語仕様および Atom の記述方法について述べる．

```
 1  extern plus();
 2
 3  main() {
 4    local fib[0..20];
 5
 6    fib[0]  = 0;
 7    fib[1]  = 1;
 8    fib[n] := plus(fib[n-2], fib[n-1]);
 9
10    return fib[19];
11  }
```

```
1 void *plus(void *a, void *b){
2   return (void *)((int)a + (int)b);
3 }
```

Mol　　Atom

図 2　Fibonacci 数を計算する Molatomium コード
Fig. 2　Molatomium code to calculate Fibonacci numbers.

2.1 Mol

Mol は次の点で特徴づけられる．

C 風の構文． 開発者が感じるギャップを少なくする．

宣言的並列性． 開発者は Atom 間のデータフローだけを考えればよく，厄介な同期を気にする必要がない．

単一代入． 変数への代入を 1 回限りと制限することで，開発者が並列プログラミングする際に混入しがちなバグを回避できる．

Mol から呼び出す Atom はすべて最初に宣言する必要がある．図 2 の Mol コード 1 行目は，この Mol コードで用いる Atom `plus` の宣言をしている．

Mol では関数を定義することが可能であり，たとえば 3 行目にある `main()` のように書く．並列実行される単位である Atom とは異なり，Mol 内の関数は逐次的に実行される．Mol コードには 1 つの `main` 関数があり，C プログラムのように Molatomium プログラムのエントリポイントとなる．引数をとる `main` を書くことで，コマンドラインからの引数を受け取ることも可能である．Atom および Mol の関数を呼び出す方法は C で関数呼び出しをする方法と同じである．

Mol はグローバル変数を持たず，関数に対してローカルな，レキシカルスコープを持つ変数のみが利用できる．4 行目では，main 内で用いるローカル変数 fib を宣言している．変数宣言にはキーワード `local` を用いる．多次元配列を宣言することができ，その範囲を `from..to` のように指定する．C と異なり，`from` と `to` の値に負の整数をとることができ，たとえば画像処理などで端の処理を簡潔に記述することが可能になっている．

Mol の変数はデータフロー変数として扱われる．すなわち，ある変数の値を読み出す命令があったときに，その変数の値が未決定であるならば，データフローグラフに依存関係が登録され，変数の値が決定されてから命令が継続実行される．また，変数は型付けされていない．このため，Atom の引数および返り値に，C/C++ の任意の型を用いることができる．Mol で変数への代入にリテラルとして用いることができるのは，現在のところ整数のみである．

データフローの表現は，変数への単一代入を用いたパターンマッチとして記述される．変数への代入を記述するには 2 つの方法があり，6–7 行目にあるような即時代入文と，8 行目にある遅延代入文である．即時代入文は，C における代入文と同じ意味を持つ．すなわち，Mol プログラムの実行がその行を通った際に右辺値が計算され，左辺値への代入が行われる．一方で遅延代入文には “`:=`” という代入演算子を用いて記述する．右辺値は即時には実行されず，むしろ左辺値が必要になった時点で展開される．すなわち，10 行目で必要とされる `fib[19]` の値を計算するために `fib[19] := plus(fib[17], fib[18])` が展開され，そこで必要とされる `fib[17], fib[18]` を計算するために … というようにデータフローの依存関係グラフが構築されていく．`plus(fib[17], fib[18])` の Atom 呼び出しは，`fib[17], fib[18]` の値が計算されるまで実行が遅延される．一方，たとえば `fib[2] = plus(fib[0], fib[1])` の計算が終わるまで実行が進んだ状態を考える．単純に Fibonacci 数を再帰的に計算するプログラムを C で書くと，これ以降も `fib[2]` の値が必要になった時点で `fib[2] = plus(fib[0], fib[1])` が実行されるが，Molatomium においてはデータフローと単一代入のためにメモ化が行われるため，これ以降 `fib[2]` を計算するための Atom 呼び出しは実行されない．

データフローの定義におけるパターンマッチは上から順に評価され，最初にマッチしたものがその配列の要素として計算される．配列の添字に用いるシンボル `n` は，パターンマッチのために用いるものであらかじめ宣言する必要がなく，その文に局所的な変数である．現在のところ，パターンマッチに用いることができるのは，配列の要素を指定するための整数，あるいは前述の局所的な変数のみである．後述するように，右辺値において三項演算を用いることで複雑なパターンを記述することも可能であるが，より自然な記述方法を追求する必要がある．

定義されたデータフローを解釈した結果，依存関係のないことが判明している複数の Atom は並列に実行可能である．配列の個々の要素が並列に計算されることがデータ並列に，異なる配列の値を計算する複数の Atom が並列に実行されることがタスク並列に対応する．

```
 1  extern run_edge();
 2  extern run_ccr();
 3  extern run_pocs_even(), extern run_pocs_odd();
 4  extern get_loop_count();
 5  extern get_frame(), put_frame();
 6
 7  main()
 8  {
 9    frame(frame_no, input)
10    {
11       local loop;
12       local edge[-1..9] outside(0);
13       local pocs[-1..9] outside(0);
14       local ccr[-1..9]  outside(0);
15       local line[0..9];
16
17       loop     = get_loop_count();
18       edge[i] := run_edge(input, i);
19       ccr[i]  := run_ccr(edge[i-1], edge[i], edge[i+1], i);
20       pocs[i] := (i%2==0)
21         ? run_pocs_even(ccr[i-1], ccr[i], ccr[i+1], loop, i)
22         : run_pocs_odd(pocs[i-1], ccr[i], pocs[i+1], loop, i);
23
24       return &line;
25    }
26
27    sync for(i in [0..30]) {
28      put_frame(frame(i, get_frame());
29    }
30  }
```

図 3　超解像アルゴリズムを記述する Mol コード
Fig. 3　Mol code to describe parallel Super Resolution algorithm.

これまでに述べてきた構文に加え，Mol は `for, while, if` といった制御構文も持つ．たとえば，**図 3** は，超解像[1] アルゴリズムを記述する Mol コードであるが，27 行目にて `for` 文を用いてフレームごとの超解像処理を 30 回反復することを記述している．`for` 文の繰返しには，変数宣言で見たような範囲指定記述を用いることができる．また動画像処理においては，出力するフレームの順序が守られていることが必要なので，`sync` キーワードを用いている．`sync` キーワードで囲まれたブロックは，ブロック内のコードを直列に実行する．ただし，ブロック内で呼ばれる関数（ここでは `frame`）の中身については，並列実行されることに注意されたい．すなわち，frame は 1 つずつ処理される一方で，その中の実際の計算は並列に実行される．

ほかに図 3 で見られる Mol の構文として，11–14 行目にある `outside` 修飾子がある．

`outside` 修飾子は，配列の宣言された範囲を超えたアクセスがあった際に返すデフォルトの値を記述するものである．これは，画像処理でよくある端の処理を単純化する．

また，20–22 行目で見られるように，三項演算子をパターンマッチに用いることができ，複雑なアルゴリズムを平易に書き下すことが可能である．24 行目で `return` 文への引数として，配列変数に `&` 修飾子を用いているが，これはその配列の要素すべての計算が終わるのを待ち，配列全体を返すということを表現している．こうすることで，バリア同期と同等な機能を実現している．

以上で見てきたように，Mol によるデータフロー記述はプログラムの並列性とメモ化を，C に似た構文を用いて簡潔に表現するものである．また，同期について考える重荷を開発者から解放する．処理系が Atom 間のデータ同期について一手に引き受けるためである．さらに，Mol のプログラムはコア数の変化に応じてスケールし，開発者は，コアがいくつあるか，個々の Atom をどのコアに割り振るか，といったことを考える必要がない．開発者はただ Mol によって並列性を記述し，それぞれのプラットフォームごとに Atom を実装するだけで，移植性とスケーラビリティを得ることができる．

2.2 Atom

Atom は CUDA [2] や OpenCL [3] の Kernel に相当する，並列実行の単位である．C/C++ を用いて記述した Atom はプラットフォームネイティブなライブラリにコンパイルされ，Mol のバイトコードとともに Molatomium ランタイムにリンクされる．Atom を C/C++ で記述することにより，これまでに最適化されてきた直列のソフトウェア資産を流用できるという利点がある．

図 2 にあるように，Fibonacci 数を計算する Molatomium プログラムの Atom 側は単に与えられた 2 つの数を加算して返すだけのものである．2.1 節で見たように，Mol には型が存在しないため，C/C++ で Atom を記述する際には引数および返値の型として `void *` を指定する．すなわち，ユーザは自身が望むどのような型を用いてデータフローを記述してもよく，大きな柔軟性がある．

Mol がプラットフォーム非依存なバイトコードとしてコンパイルされることでポータビリティを確保しているのに対して，Atom はプラットフォームごとに書く．このため，プラットフォーム固有の最適化技術，たとえば SIMD 命令を活用することができる．並列プログラミングのにおいて重要なのは，スケーラビリティやポータビリティといった全体としての性能と，個々のコアで動作する直列コードの性能とのバランスである．Molatomium は，Mol でスケーラビリティとポータビリティを確保する一方で，ユーザの最適化に関する知識を Atom のチューニングに活かすといった実用的なバランスを目指した．

3. 処　理　系

Molatomium 処理系は仮想マシンとして実装されており，Mol のバイトコードを実行し，依存関係のデータフローを管理する．それぞれのコア，スレッドに処理系が常駐し，自律的にバイトコード処理と Atom 処理とをスケジュールする点が特徴的である．

バイトコードを並列記述として用い，実行環境を仮想マシンで構築することには，以下のようなメリットがある：第 1 に，Molatomium の並列プログラムを移植するのが容易である．ターゲットプラットフォームが変わっても，開発者は Mol コードを再コンパイルする必要はない．第 2 に，プログラムを実行時の情報を用いて最適化することが可能になる．メディア処理といったプログラムは，入力データによって処理量が変動しうる．そのため，静的解析によるタスクスケジューリングでは，所望の性能が得られないことがある．

Molatomium の実行モデルは Mol のコードを基に実行時に依存関係のグラフを構築することである．依存関係のグラフには，未決定状態の変数，計算済みの変数についてのメモ化といったデータの管理が求められ，単純なスタックマシンとは異なる．こうした実行モデルを実現するには，処理系を仮想マシンとするのがシンプルな解決策である．

処理系は自身の状態と，処理系実行のためのロックとを共有メモリに置く．初期化コードは，処理系と Molatomium プログラムとをすべてのコアに配置する．

3.1 実　　装

図 4 に，処理系がどのように動作するかを示す．ロックを保持しているコアは，実行可能な Atom 呼び出しに到達するまで，Mol バイトコードを順に解釈実行する．他のコアは

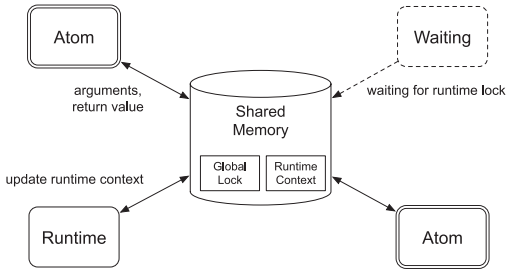


図 4　各コアの動作
Fig. 4　Each core activities.

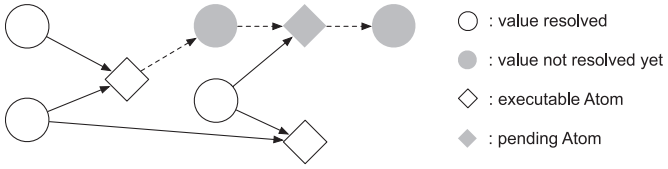


図 5　データフロー
Fig. 5　Data flow.

Atom コードを実行中であるか，処理系のロックを獲得できるのを待っている状態にある．各コアで自律的に処理系を担当する仕組みは，集中的にタスクスケジューリングをするコアを必要としないため，スケーラビリティの向上が期待できる．

バイトコードの実行は，依存関係のグラフを構築していく作業になる．**図 5** に，依存関係のグラフがどのようにつくられていくかを示す．実行可能な Atom（白い四角形）とは，引数となる変数の値がすべて決定している（白い円）状態のものである．まだ値が決定していない変数（灰色の円）がある場合，処理系はその Atom について，引数となる変数と，返り値を格納するための変数との間にリンクをはり，Atom の遅延実行のための待ち行列に入れ，実行を先送りし（灰色の四角形），バイトコード解釈を継続する．Atom 呼び出し以外のバイトコード処理においても引数の値が未決定の場合には，そのバイトコード処理を待ち行列に入れて評価を先送りにする．すべての引数の値が決定している Atom 呼び出しに到達すると，処理系はロックを外し，Atom コードの実行に移る．これは，処理系のロックを獲得しようと待っている他のコアに，処理系の実行を引き継ぐことを意味する．Atom を実行しているコアが処理を完了すると，返り値を格納するために関連づけられている変数に返り値を代入し，処理系のロックを獲得するための待ち状態に入る．

待ち行列に空きがある限り，データの依存関係が解決していない処理を先送りにしつつ，依存関係を解消するための Atom 呼び出しを並行して実行していくことで，全体の並列実行性の向上を見込む．高い並列性を獲得するためには，各コアで Atom の処理が実行されていて，処理系のロックを待つ時間が小さくなるよう，Atom の計算量が十分大きくなるよう設計することが開発者に期待される．

3.2　Cell/B.E. における実装

Molatomium 処理系のポータブルな実装例として，ヘテロジニアスマルチコアプロセッサである Cell/B.E. の実装について述べる．

処理系と Atom は，どちらも SPE 上で動作し．PPE は単に初期化と終了を行うのみである．SPE の LS は 256 KB しかないため，処理系が占めるメモリ容量は十分に小さい必要がある．現在の実装では，動的なメモリ管理サブシステム込みで 70 KB を占めるのみであり，ユーザのプログラムは残りの 186 KB すべてを使うことができる．

遅延実行のための待ち行列は LS の小ささにともなって小さくせざるをえない．待ち行列に空きがなくなると，処理系を持っている SPE は Atom 処理を行っている SPE が Atom を実行し終え，データ依存が解決し，待ち行列が空くのを待つしかない．これは並列性を阻害する要因である．この問題を避けるためには待ち行列をできるだけ大きくするのが良いが，一方で Atom の処理内容によっては LS を多く使用できればできるほど処理速度が向上するものがあり，トレードオフとなることがある．そのためには自動的に待ち行列の大きさを最適な値に調節する，4 章で述べるツールでプロファイル情報を提示することでユーザに最適化してもらう，などの方法が考えられる．

Cell/B.E. と典型的な x86 によるマルチコア環境との間にある主な違いは，Cell/B.E では SPE 間でデータのやりとりのために DMA を開発者が明示的に実行する必要がある点である．処理系の状態とユーザのプログラムは共有メモリに置かれ，SPE は各々が LS にコピーを持つ．ロックはアトミックな DMA 命令を共有メモリに発行することで実現し，処理系の状態はアトミックでない DMA を用いる．DMA のコストは Atom の処理量が十分に大きければ隠蔽可能であるし，処理系の状態を差分だけ転送することでさらに削減可能である．

4. ツ　ー　ル

2 章で述べたように，Molatomium は C 言語に親しんだ開発者が容易に利用できるよう配慮して設計している．それでもなお，新しいプログラミング言語の習得に壁を感じられてしまうことも事実である．そこで，Molatomium によるプログラミングの理解を促進するために IDE のようなグラフィカルな開発環境を用意した．**図 6** にグラフィカルエディタを示す．

開発者は四角形で表現される Atom を，丸で表現される変数とつなぐことでデータフローを記述し，直感的にプログラミングを行うことができる．同じ配列の別要素を計算する式を定義するには，データフローのグループを矢印でつなぐことで表現する．Atom の四角形を展開すると，Atom の C/C++ コードが編集できる．

図 7 は，グラフィカルデバッガである．図 6 のエディタで記述したデータフローを展開した図の上で，Atom や変数に対してブレークポイントを張り，計算が正しく行われている

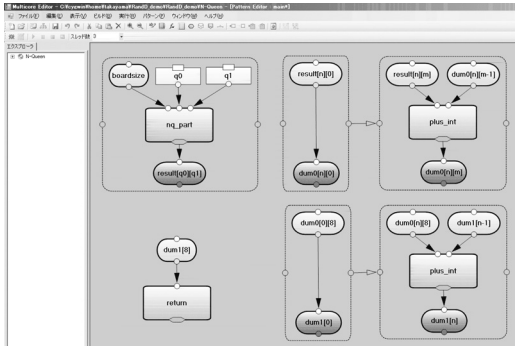


図 6　IDE エディタで N-Queen
Fig. 6　N-Queen in IDE Editor.

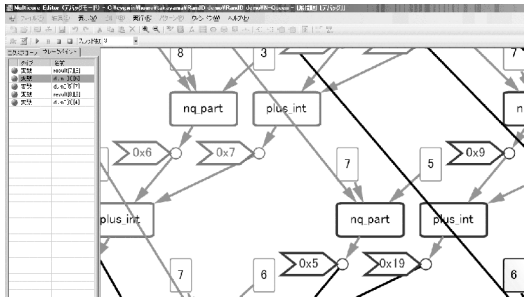


図 7　IDE デバッガで N-Queen
Fig. 7　N-Queen in IDE Debugger.

かを確認することができる．また，実行中の Atom はコアごとに色分けされており，どのコアでどの Atom が実行され，どの変数が決定されたかを見てとれるようになっている．これは，並列性が十分に出ているかを確認するためにも用いることができるものである．また，Atom を指定してステップ実行を行うことができ，開発者が自分の書いた並列プログラムがどのように動いているかを理解するのに役立つ．ステップ実行は Mol のコードだけでなく，C/C++ で書かれた Atom の中にもシームレスに入ることができ，言語をまたいだデバッグができるようになっている．

並列プログラムの解析とデバッグには，可視化が有効であると考える．ダイナミックに三次元のグラフ構造を生成する Ubigraph[4] を処理系に組み込み，ユーザのプログラムがどのように動いているか直感的に把握できるようにした．

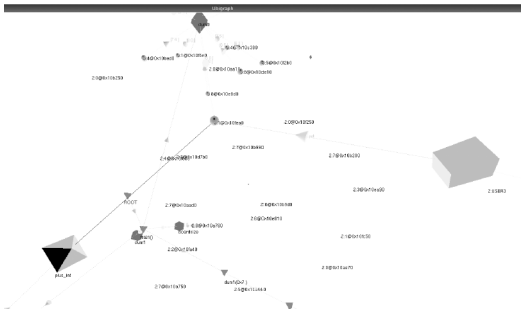


図 8　N-Queen の実行を三次元可視化
Fig. 8　3-D visualization of N-Queen.

図 8 は，Molatomium で書いた N-クイーン問題が実行される様子を可視化したものである．左下にある濃い色の矩形は実行中の Atom であり右にある矩形がその実行結果を待つ Atom である．データの依存関係は矩形間をつなぐ矢印で表現されており，どの変数がどんな値を持っているかという情報が辺上のラベルとして表示されている．実行中の Atom，すなわち濃い色の矩形が多いほど，プログラムが並列に動いていることになる．期待する並列度が出ているかどうか直感的に目で見て把握できることが，可視化することの利点である．

5. 評　　価

Molatomium の処理系を，いくつかの x86 POSIX プラットフォーム（Linux，Mac OS X，Cygwin）および Cell/B.E. と SpursEngine に実装した．アプリケーションとしては，N-クイーン問題などのトイプログラム，MPEG-2 デコーダや H.264 デコーダ，超解像処理といったメディア処理アルゴリズムを選んだ．

図 9 に N = 14 のときの N-クイーン問題の，**図 10** に超解像処理におけるスケーラビリティを示す．それぞれ x86 と Cell/B.E. の両方において評価し，Molatomium を用いずに手でプログラムを並列化したものと Molatomium を用いたものとを比較した．

x86 の評価環境は物理 4 コアの Intel® Core™ i7 でハイパースレッディングを有効にしたものであり，OS は Ubuntu Linux 8.04 を使用した．Cell/B.E. については Sony Playstation 3 を用いており，PPE が 1 コアで SPE が 6 コア利用可能なもので，OS には Fedora

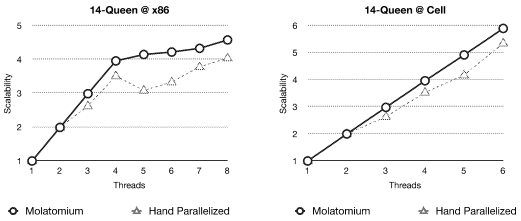


図 9　スケーラビリティの評価：14-クイーン
Fig. 9　Evaluation of scalability: 14-Queen.

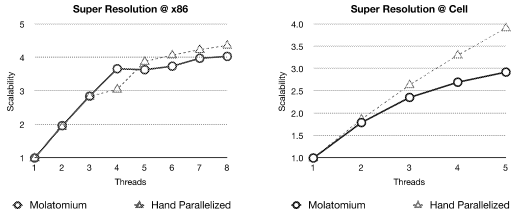


図 10　スケーラビリティの評価：超解像処理
Fig. 10　Evaluation of scalability: Super Resolution.

Core 9，SDK に IBM SDK for Multicore Acceleration 3.1 を使用した．どちらのアルゴリズムについても，x86 におけるスケーラビリティが 4 コアまでしか出ていないのは，この評価環境では物理コア数が 4 のためである．

14-クイーン問題の評価結果は，Molatomium の動的なタスクスケジューリングの効率の良さを示すものである．手で並列化した 14-クイーン問題は静的にタスクをコアに割り当てている．x86 と Cell/B.E. の両プラットフォームにおいて，手で並列化したものよりも Molatomium 版でスケーラビリティが出ていることが分かる．

超解像処理の評価は，Molatomium 処理系のオーバヘッドを明らかにするものである．手で並列化したものは，Cell/B.E. における並列プログラミングに十分に習熟しているエキスパートによる実装であり，超解像処理のために高度に最適化された動的タスクスケジューラを持つ．一方で，Molatomium のタスクスケジューリング方法は汎用的なものである．加えて，超解像処理における Atom の処理量は N-クイーンのものよりも軽量であるため，処理系のオーバヘッド，すなわち処理系がバイトコードを解釈実行し，データの依存関係グラフを構築していくのに要する処理に要する時間が相対的に増加する．

このオーバヘッドを削減するためには，Atom コードを束ねることで処理量を増やして Atom と処理系の間で処理量のバランスをとることになる．このために，4 章で述べた開発環境において，実行した Atom のプロファイルを基に開発者に処理量のバランスをとるためのヒントを与えるツールを作成した．また，Atom のプロファイルを実行時にとりながら，最適な Atom の処理量になるように制御する粒度最適化を，処理系に組み込むことが考えられる．

いずれのアプリケーションも，4 章で述べた x86 で動く開発環境を用いて設計，実装，デバッグした．x86 で開発したアプリケーションを Cell/B.E.，SpursEngine に移植するのはこれまでになく容易であった．開発者は単にそれぞれのプラットフォームに対する Atom コードの最適化について考えるだけでよく，並列化については Mol を記述するだけになったことで，開発期間が最大で半分に短縮された．

6. 関 連 研 究

SMP における並列プログラミングモデルの代表的なものとして，Cilk [5] がある．Cilk では `cilk, spwan, sync` といったキーワードを既存の C プログラムに加えることで直列プログラムのタスク並列化を行うことができる．すなわち，C の動作意味を保持したまま，プログラムの並列化を行えるという利点がある．一方で，プログラム中のどこを spawn すればよいのか，粒度はどの程度が最適なのかといったことを考える必要があり，またユーザが明示的に sync でバリア同期を意識する必要がある．Molatomium においても，プログラム中のどの部分を Atom 化すればよいのか，そしてその粒度について考える必要は残る．我々がとるアプローチは，Atom をできるだけ細粒度に書いてもらい，その粒度についてはランタイムが実行時に最適化するというものである．実行時のプロファイルによって細かすぎると判明した Atom は，データフローで結び付けられている他の Atom と動的に結合され，適切な粒度になるという姿を目指している．

Intel® Concurrent Collections for C/C++ (CnC) [6] は，自動的にデータフローを解決するプログラミングモデルを持つという点で，Molatomium に近い．CnC には *tag* という構成要素があり，イベント駆動による並列処理を記述することが可能になっているという点で Molatomium よりも高機能であるといえる．一方で，template を駆使した C++ による記述を行う必要があるために記述量が多く，また C プログラマから見て敷居が高い．

MARS [7] は，Cell/B.E. における並列プログラミングの難しさに対して，タスクキューモデルというアプローチで取り組んでいる．MARS のタスクキューモデルは，タスクを登録するためのキューをすべての SPE が共有し，各 SPE で動作するスケジューラがタスク

を並列に実行する．個々の SPE が自律的にタスクのスケジューリングを行うという点で Molatomium と通じるところがある一方で，MARS はタスク間の同期を明示的に記述する必要があるという点で Molatomium より低いレイヤの技術と見ることができる．そのため，Molatomium を MARS のランタイムの上で動かすことも考えられる．タスクキューモデルには，ほかに Grand Central Dispatch [8] や OpenCL [3] のタスク並列部分などがある．

OpenMP [9] は，多くのコンパイラがすでにサポートしている並列プログラミング技法で，要素間に依存関係のない配列を計算するループといった単純なプログラムを，並列化のためのプラグマを挿入するだけで容易に並列することができる．一方で Molatomium のデータフローによる並列プログラミングモデルは，超解像といった高度な画像処理アルゴリズムで見られる，データ依存関係が複雑で単純なループ並列化で実現できない問題も解決するものである．

宣言的並列性については，CTMCP [10] に詳しい．Concurrent Prolog と Concurrent Haskell は宣言的並列性に基づいたプログラミング言語の例である．

7. おわりに

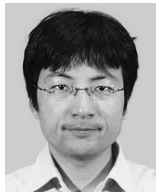
プロセッサのマルチコア化はユーザに省電力と性能向上を約束する一方で，プログラミングモデルの未成熟さにより開発者に苦難を強いている．Molatomium は開発者から厄介な同期問題を解放し，彼らの本当の仕事，すなわちアルゴリズムの開発と高速化に集中することを支援するものである．Molatomium は SpursEngine 搭載 PC や Cell/B.E. 搭載 TV 内における超解像，フレーム補完，圧縮歪み除去，各種コーデックといった各種アプリケーションを並列化する際の基盤技術として，すでに製品化されている実績がある．これは，Molatomium の実用性を示すものである．今後の課題として，Mol の言語によるコア間のデータ転送支援，メモリ階層に対する最適化支援，ランタイムの GPU 対応，およびランタイム自体も並列化することによるメニーコアにおけるスケーラビリティの確保，などがあげられる．

参考文献

1) Ida, T., Matsumoto, N. and Isogawa, K.: Reconstruction-based Super-resolution Using Self-congruency of Images, IEICE technical report, Image engineering, Vol.107, No.380, pp.135–140 (20071206).
2) NVIDIA: CUDA. http://www.nvidia.com/cuda
3) Khronos: OpenCL. http://www.khronos.org/opencl
4) Veldhuizen, T.L.: Dynamic Multilevel Graph Visualization, *Eprint arXiv: cs. GR/07121549* (2007).
5) Blumofe, R., Joerg, C. and Kuszmaul, B.: Cilk: An efficient multithreaded runtime system, *ACM SIGPLAN Notices* (1995).
6) Intel: Concurrent Collections for C/C++. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/
7) Levand, G.: MARS, Multicore Application Runtime System (2008). ftp://ftp.infradead.org/pub/Sony-PS3/mars/
8) Apple: Grand Central Dispatch. http://libdispatch.macosforge.org
9) Dagum, L., Menon, R. and Inc, S.: OpenMP: An industry standard API for shared-memory programming, *IEEE Computational Science & Engineering* (1998).
10) Roy, P.V. and Haridi, S.: *Concepts, Techniques and Models of Computer Programming*, MIT Press (2004).

(平成 21 年 7 月 10 日受付)
(平成 21 年 10 月 19 日採録)

高山　征大
2004 年京都大学大学院情報学研究科修了．同年株式会社東芝入社．並列プログラミング，可視化等に興味を持つ．



境　　隆二（正会員）
1990 年京都大学理学部卒業．同年株式会社東芝入社．コンパイラ・ミドルウェアの開発に従事．速いソフトを早く実装することに興味を持つ．

加藤　宣弘（正会員）

1988 年京都大学大学院修士課程修了．同年株式会社東芝入社．組み込みシステム向けソフトウェアの開発に従事．

島田　智文（正会員）

1984 年早稲田大学大学院理工学研究科物理学および応用物理学専攻修士課程修了．同年株式会社東芝入社．コンピュータ，組み込みシステムの基盤ソフトウェアに関する研究・開発に従事．