Vol. 42　No. SIG 12(HPS 4)　情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム　Nov. 2001

プロセスネットワークを宣言的に記述する並列言語

大 野 和 彦[†]　山 本 繁 弘[†]
岡 野 孝 典[†,☆]　中 島 　 浩[†]

非数値分野のプログラミングでは非定型・動的なデータ構造が多用されるため自動並列化が困難であり，従来より様々な並列化ライブラリや並列言語が提案されてきた．しかし，一般的に実行効率の高いものは低レベルな記述が必要であり，抽象性の高いものは効率的な実装が難しい．そこで我々は，記述のしやすさと実行効率を両立させることを目標に，並列言語 Orgel の研究開発を行っている．Orgel では，実行単位であるエージェントが並行/並列に動作し，抽象通信路であるストリームを介してメッセージを送りあう．同種のモデルに基づく既存言語と異なり，Orgel ではこのプロセスネットワークを宣言的に記述する．この結果，同期/通信タイミングによるバグを防ぎ，プログラム記述を容易にしている．また，コンパイル時に実行モデルの構造が分かるため，静的解析により強力な最適化を施すことができる．現在，共有メモリ型並列計算機上に Orgel 処理系を実装済であり，これを対象とした性能評価を行った．その結果，逐次実行のオーバヘッド・並列実行での速度向上率ともに，直接 Pthreads ライブラリを用いる場合と比較して遜色なく，他の動的要因の大きい並列言語と比べて効率が良いことが示された．また，並列化のためのコード変更コストについても Orgel が優位であり，ランタイムライブラリの結合による実行バイナリサイズの増加も，Pthreads 版の 2 割〜6 割増程度にとどまった．

A Parallel Programming Language Based on Declarative Process Network Models

Kazuhiko Ohno,[†] Shigehiro Yamamoto,[†] Takanori Okano[†,☆] and Hiroshi Nakashima[†]

Automatic parallelization is much difficult in non-numerical field, because irregular and dynamic data structures are frequently used. Therefore, many parallelizing libraries and parallel programming languages have been proposed. However, the efficient systems tend to force low-level specifications to the programmers. And the highly abstracted systems are difficult to implement effiently. So, we are developing a parallel programming language named *Orgel*, which aims for both efficiency and easiness. In the execution model of Orgel, the execution units called *agents* run in concurrent/parallel and send messages via abstract channels called *streams*. Unlike many other languages based on the similar models, the process network model is declaratively specified. This feature prevents timing bugs and simplifies parallel programming. Furthermore, the program can be strongly optimized using static analysis because the execution model is known at compile time. We have implemented Orgel on shared-memory multiprocessors. The result of evaluation shows that both the overhead in sequential execution and the speedup in parallel execution can match with the programs using Pthreads library. Parallelization was much easier using Orgel, and the increase of the executable size caused by linking Orgel runtime library is only 23–65% larger compared to the Pthreads version.

1. はじめに

我々は，非数値並列処理を記述するためのプログラミング言語の研究を行っている．配列など定型的データをループで処理することの多い数値処理と異なり，この分野では非定型的・動的なデータ構造が多用され，プログラムの制御構造にも再帰が多く現れる．このため，数値処理分野で研究が進んでいるような自動並列化は非常に困難であり，ユーザが明示的に並列性を記述したプログラミングを行うことが多い．具体的なプログラミング手法としては，C など逐次の手続き型言語上で並列化ライブラリを使う方法や，明確な並列実行セマンティクスを持つよう設計された新しい言語を用いる方法などが使われている．

† 豊橋技術科学大学
Toyohashi University of Technology
☆ 現在，株式会社日立システムアンドサービス
Presently with Hitachi Systems & Services, Ltd.

前者の例としては PVM[1]，MPI[2]のような通信ライブラリや POSIX スレッドライブラリ（以下 Pthreads ライブラリ）[3]などがある．これらを用いる方法は，ユーザにとって逐次言語からの移行が比較的容易であり，細部を自由かつ効率的にプログラムできる．その反面，低レベルな記述が必要であり，不慣れなユーザにとっては通信や同期のタイミングバグを生じやすい．とくに非数値分野では，不定回数の通信や異なるメッセージが非同期に到着する通信が頻繁に用いられるため，誤りのない並列プログラムを書くには高いスキルを要求される．

一方，後者の例としては KL1[4]や ABCL[5]などがあげられる．これらは並列記述のための抽象度の高い記法を導入することで，前者で見られる並列化にともなうバグを防いでいる．しかしながら，通信や同期をランタイム内で自動的に処理するため，そのオーバヘッドで実行速度が低下することが多い．また，こうした言語の記述方法は C などの普及した逐次言語とは異なる点が多いのも，逐次言語からの移行を妨げている．

そこで我々は，両者の利点をあわせ持つ並列言語 Orgel を提案し，研究を行っている[6),7]．Orgel では，並行並列に動作するエージェントがストリームと呼ばれる抽象通信路を通して互いにメッセージを送りあうというスタイルで，プログラムを記述する．エージェント内部の動作を逐次言語で効率的に記述する一方で，プロセスネットワークの記述には宣言的な記法を採用している．これによって，ユーザにとって見通しの良い並列プログラミングを可能にすると同時に，処理系にとって詳細な静的解析を可能にし，強力な最適化を施すことを狙っている．

現時点で，逐次/共有メモリ並列環境で実行可能な Orgel 処理系（以下，共有メモリ版処理系）が実装済であり，現在は分散メモリ環境への対応を進めている．また，上記の言語特性を生かしたデバッギング手法の研究[8]なども進めている．

以下，2 章で Orgel の言語仕様を説明し，3 章で記述性について考察する．4，5 章では共有メモリ版処理系の実装方法と性能評価の結果を述べる．6 章で関連研究に触れ，最後に 7 章でまとめを行う．

2. 並列言語 Orgel

2.1 プログラミングモデル

Orgel の実行モデルは，エージェントと呼ばれる複数の実行単位をストリームと呼ばれる抽象通信路で結んだプロセスネットワークで表現される．これらのエージェントは並行並列に動作し，ストリームを通してメッセージを交換しながら，各々の処理を行う．

他のマルチエージェント/アクティブオブジェクト型言語[5),9)~12]でも同様なモデルを用いているが，多くの場合はプロセス生成やネットワーク接続を操作的に記述する．このため非常に柔軟な記述が可能である反面，実行時のある局面におけるプロセスネットワークの状態が予測しにくく，生成・通信タイミングや接続対象の誤りを生じやすい．

これに対して Orgel では，プロセスネットワークの構造を宣言的に記述することを特徴とする．すなわち，構成要素であるエージェントやストリームのインスタンスを静的に定義された変数で表し，それらの結合方法も接続宣言文により静的に記述する．

この結果，操作的な記述を行う言語に比べてモデルの形状が明確になり，バグの混入を防ぐと同時に静的解析を容易にして効果的な最適化ができるようになっている．一方で，動的操作を禁じることにより記述力には制限が生じ，たとえば OS などシステムプログラムの記述は非常に困難である．しかしアプリケーション記述では，非数値分野でもネットワーク構造自体は定型的であったり，非定型的でも動的に変化しなかったりするものが数多くあるため，本言語のアプローチは十分有用であると考えている．また，この分野で多用される木探索型のプログラムも，2.6 節で述べるように再帰的な接続を定義することで，Orgel で扱うことができる．

2.2 言語の概要

Orgel は C に対し，以下の変更を加えたものになっている．

(1) ストリーム/メッセージ/エージェントを表す型と，それを宣言する構文を追加．
(2) メッセージの作成/送信/参照やエージェント終了の構文を追加．
(3) エージェントメンバ関数を追加．
(4) 大域変数を禁止し，エージェントメンバ変数を追加．

(1) については，C における構造体や列挙を用いた型定義と同様に，ユーザが構造を記述して型名を定義する．以下，これらの型をそれぞれストリーム型，エージェント型などと総称し，特定の型については ‘`main` 型’ もしくは ‘`main` エージェント型’ のように表記する．

次節より，Orgel の言語仕様について詳しく述べる．例として，**図 1** のようなマスターワーカー型のプロセスネットワークを考える．マスターは最初に，すべてのワーカーへあるていど大きなデータを放送する．

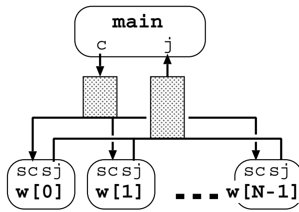

図 1 マスターワーカー型のプロセスネットワーク
Fig. 1 Process network of master-worker program.

```
stream Scontrol {
    Mdata(char data[M]:in);
}
stream Sjob {
    Mrequest(Mjob job:out);
    Mjob(int param:in);
    Mresult(char data[M]:in);
}
agent Aworker(Scontrol sc:in, Sjob sj:out){
    char d[M], nd[M];
    initial { taskstate = NO_TASK; }
    dispatch(sc){
        Mdata(d): { taskstate = HAS_TASK; }
    }
    task {
        Mjob job;
        sj <== Mrequest(job);
        job ?== Mjob(p);
        p と d から nd の値を作成;
        sj <== Mresult(nd);
    }
}
agent main(){
    char d[M];
    Aworker w[N];
    Scontrol c;
    Sjob j;
    connect self ==> c ==> w[].sc;
    connect self <== j<== w[].sj;
    initial {
        初期データ d を作成;
        c <== Mdata(d);
    }
    dispatch(j){
        Mrequest(job): job = Mjob(make_param());
        Mresult(rd): { if (rd が d より良い){
                          d を更新;
                          c <== Mdata(d);
                        }
                     }
    }
}
```

図 2 マスターワーカー型のプログラム（Orgel 版）
Fig. 2 Master-worker program (Orgel version).

ワーカーはマスターに仕事の要求を送って処理のパラメータをもらい，それに従ってデータを処理し，結果をマスターに送る．その後，再び仕事を要求し，処理を繰り返す☆．マスターは受け取った処理結果を検査し，元のデータより良い値が得られればデータを更新し，再び全ワーカーに放送する．なお，問題の簡略化のため，終了条件についてはここでは考えないことにする．Orgel 版のプログラムを**図 2** に示す☆☆．

```
agent AgentType(
   [StreamType StreamName: Mode [,...]]
){
  メンバ変数宣言
  メンバ関数プロトタイプ宣言
  接続宣言
  initial Initializer;
  final Finalizer;
  task TaskHandler;
  dispatch (StreamName){
    MessageType: MessageHandler;
       ...
  }
}
```

図 3 エージェント型の宣言
Fig. 3 Declaration of an agent type.

2.3 エージェント

エージェントは，内部状態を表すメンバ変数，自身の処理を行うコード，受信した各メッセージに対する応答コードから成る．

ある機能を持つエージェントを作成するには，まず**図 3** のような形式でエージェント型を宣言する．

エージェント型名 *AgentType*の直後に列挙されるストリームの並びは，このエージェント型が持つ入出力ストリームである．*StreamName*は仮引数であり，2.6 節で説明する接続宣言により実際のストリームと結合される．*Mode*は in または out を指定し，そのストリームが入力/出力のいずれであるかを示す．

メンバ関数は，このエージェント型からのみ呼び出せる関数である．定義はエージェント宣言の外で従来の C の関数と同様に行うが，関数名を *AgentType*::*FunctionName*の形式で指定する．メンバ変数は，実際に生成された個々のエージェントごとにメモリ領域が確保され，エージェント宣言内のコードと，このエージェント型のすべてのメンバ関数をスコープとする．

`initial`，`final`，`task`，`dispatch` には，実際の処理を行うハンドラを定義する．ハンドラには C の単文もしくはブロックが記述できる．この部分は基本的に

☆ 結果送信と次の仕事要求を同時に行った方が効率が良いが，処理を分かりやすくするため，ここでは別々に行うものとする．
☆☆ Orgel のエージェント/ストリーム/メッセージ型の命名規則は C の識別子の場合と同じであるが，ここでは視認性を良くするため，それぞれ大文字の A，S，M を接頭辞としている．

```
stream StreamType {
    MessageType [(Type Arg :Mode [, ...])];
      ...
}
```

図 4　ストリーム型の宣言
Fig. 4　Declaration of a stream type.

C の逐次コードとなるが，後述するメッセージ送信/参照などの言語拡張を使用できる．*Initializer*，*Finalizer*にはそれぞれこの型のエージェントの生成・破棄時の処理を，*TaskHandler*にはメッセージ処理を行っていないときに実行する処理を記述する．後者を実行するかどうかは，予約変数 `taskstate` の値（`NO_TASK`，`HAS_TASK`）を変更することで制御する．`dispatch` には，このエージェント型の入力ストリームに対し，受信可能なメッセージを列挙し，各々への応答処理を記述する．入力ストリームが複数ある場合は各々について `dispatch` を記述すると，書かれた順に受信を試みる．

エージェントを終了させるには，そのエージェント自身がハンドラ内のコードで `terminate` 文を実行する．

図 2 の例で定義された `Aworker` 型のエージェントは，入力ストリーム `sc` より処理データを受け取る．メッセージが来ないときは `task` ハンドラで `main` エージェントに仕事を要求し，返信されたパラメータに従った処理を行う，という動作を繰り返す．

2.4　ストリーム

ストリームは KL1 のストリーム通信モデルを元にした，抽象的な通信路である．ストリームは方向を持ち，両端に 1 個以上のエージェントが接続される（2.6 節参照）．受信端に複数接続された場合はそれらへのマルチキャストとなり，送信端に複数接続された場合は，各エージェントからのメッセージが非決定的にマージされる．KL1 とは異なり，Orgel のストリームは型を持ち，それぞれのストリーム型は宣言時に指定された型のメッセージだけを受け付ける．

エージェントと同様に，ストリームを使用するには，まず**図 4** のような形式でストリーム型を宣言する．この宣言は，ストリーム型 *StreamType*で送信可能なメッセージ型を列挙したものであり，同時にプログラム中で使用するメッセージ型の宣言にもなっている．

メッセージは関数形式をとり，メッセージ識別子となるメッセージ型名 *MessageType*，および引数 *Arg*の型 *Type*や入出力モード *Mode*（`in`，`out`）を列挙する．たとえば図 2 のプログラムで，`Sjob` 型のストリームは `Mrequest`，`Mjob`，`Mresult` 型のメッセージのみを受け付ける．メッセージの詳細は，2.7 節で説明する．

```
connect [A0.S0 Dir0 ] S Dir1 A1.S1;
```

図 5　接続宣言文
Fig. 5　Connection declaration.

2.5　エージェントとストリームの生成

エージェントやストリームのインスタンスを生成するには，親となるエージェント型の宣言内でエージェント/ストリーム型のメンバ変数を定義する．

あるエージェント型 A_i でこれらの型のメンバ変数を定義すると，A_i のインスタンス生成と同時に各メンバ変数に対応するインスタンスが自動的に作られる．

図 2 の例では，`main` 型のエージェントが 1 つ生成されると，`AWorker` 型のエージェントが N 個，`Sdata` 型と `Sjob` 型のストリームがそれぞれ 1 個，自動的に生成される．

プログラム起動時には `main` 型のエージェントが 1 つ生成され，各エージェントやストリームは `main` を起点として次々に自動生成される．このため，操作的なインスタンス生成を行う言語で起こりうる，生成前に呼び出してしまうといったタイミングバグは生じない．

実際にはこの自動生成は一度には起こらず，初期状態で実行可能な `task` ハンドラを持たない☆エージェントについては，最初のメッセージが送られるまで生成が遅延される．2.6 節で述べるように，この性質を利用して再帰的なプロセスネットワークを記述できる．

また，各インスタンスの配置や実行 PE は，処理系と OS により決定される．このため，実行環境が共有メモリ/分散メモリのいずれであるかや利用可能な PE 台数にかかわりなく，同じ Orgel プログラムを実行することができる．

2.6　エージェントとストリームの接続

エージェント型宣言内に**図 5** の形式の接続宣言文を記述することにより，エージェントの仮引数とストリームの接続関係を静的に指定する．*A0*，*A1*，*S*はそのエージェント型の中で定義されたエージェント/ストリーム型メンバ変数であり，*S0*，*S1*はそれぞれのエージェント型で宣言された仮引数である．*Dir0*，*Dir1*には，演算子`<==`または`==>`によりデータフローの方向を指定する．

*A0*や *A1*の代わりに予約語 `self` を指定すると，接続宣言を行うエージェント自身の仮引数をストリームに接続する．さらに，仮引数を省略して `self` のみを

☆ task ハンドラ自体が定義してあっても，initial ハンドラで taskstate の初期値を NO_TASK に設定してあれば，いずれかのメッセージハンドラでこの値を HAS_TASK に変えるまで task ハンドラは実行されない．

指定することで，仮引数を介さずにストリーム型メンバ変数を直接，送受信の対象にできる．後者の方法では子エージェントとの通信インタフェースが自身の引数に現れないため，カプセル化されたプロセスネットワークを作成することができる．

1つのストリーム端に複数のエージェントを接続する場合には，同じストリーム変数に対して複数の接続宣言を行う．たとえば以下の例では，a と c の仮引数 o からのメッセージがマージされて，b，d の仮引数 i へマルチキャストされる．

```
connect a.o ==> s ==> b.i;
connect c.o ==> s ==> d.i;
```

また，上記の例は以下のように図 5 の *A0.S0* と *Dir0* の部分を省略した記法を用いることで，ストリームとエージェントの関係を強調した分かりやすい記述とすることもできる．

```
connect s <== a.o;
connect s <== c.o;
connect s ==> b.i;
connect s ==> d.i;
```

ある仮引数についてはたかだか 1 つの接続宣言しか記述できない．仮引数への接続を行わなかった場合は，in モードならメッセージが到着することのないストリームとして振る舞い，out モードなら送信されたメッセージは即座に破棄される．接続が静的であるため，このような非接続引数の存在はコンパイラが警告できる．

配列型のメンバ変数や仮引数を使うと，1 対多や多対多の接続を簡潔に指定できる．この場合，接続宣言文の引数となる各識別子の後に，‘[添字式]’ の形式の配列指定子を指定する．添字式を省略すると配列全体が接続対象となる．

添字式は定数式もしくは疑似変数と呼ばれる暗黙の識別子を含む式である．前者の場合は，式の値を添字とする配列の一要素が接続対象となる．後者の場合，疑似変数は 1 つの接続宣言文全体をスコープとし，各添字式ごとに 1 つだけ含めることができる．疑似変数の値は，出現する各添字式が配列の大きさを超えないという条件を満たす，すべての整数値の集合である．この結果，疑似変数の各値について添字式がとる値により，接続対象となる配列の要素が示される．

1 つの添字式には疑似変数がたかだか 1 つしか含まれないため，疑似変数の値に関する条件は一元不等式の集合となる．コンパイラはこれを解くことで，該当する要素どうしを接続するコードを静的に生成し，条件を満たす値がない場合はコンパイル時エラーとできる．

```
connect self ==> c ==> w[0].sc;
connect          c ==> w[1].sc;
                  ⋮
connect          c ==> w[N-1].sc;
connect self <== j <== w[0].sj;
connect          j <== w[1].sj;
                  ⋮
connect          j <== w[N-1].sj;
```

図 6 図 2 と等価な接続宣言
Fig. 6 Connection declaration equivalent to Fig. 2.

```
Aworker w[N][N];
Sdata se[N-1][N-1], sw[N-1][N-1],
      sn[N-1][N-1], ss[N-1][N-1];
connect w[i][j].eo==>se[i][j]  ==>w[i+1][j].wi;
connect w[i][j].wo==>sw[i-1][j]==>w[i-1][j].ei;
connect w[i][j].no==>sn[i][j]  ==>w[i][j+1].si;
connect w[i][j].so==>ss[i][j-1]==>w[i][j-1].ni;
```

図 7 メッシュ型結合網の接続宣言
Fig. 7 Connection declaration of mesh network.

図 2 の例では，最初の接続宣言文でストリーム c の送信端には self により main 型エージェント自身を，受信端に各 Aworker 型エージェントの入力引数 sc を接続している．この結果，main 型エージェントがメンバ変数 c に送信したメッセージはすべての Aworker 型エージェントにブロードキャストされる．2 番目の接続宣言文では逆に，ストリーム j の送信端に各 Aworker 型エージェントの出力引数 sj を，受信端に main 型エージェントを接続している．この結果，各 Aworker 型エージェントが sj に送信するメッセージは，非決定的にマージされて main 型エージェントに届く．この 2 つの接続宣言文は**図 6** の宣言と等価である．

さらにいくつかの例を図 7〜図 10 にあげる．

図 7 ではメッシュ型の接続網を宣言している．各エージェントは四方位それぞれについて入出力引数を持ち，添字式を使って隣接するエージェントと接続している．この結果，**図 8** のようなプロセスネットワークが形成される．なお，この例ではメッシュ端のエージェントについては一部の引数が未接続になっている．このような場合，到着しないメッセージを待ってデッドロックしたりしないよう，ハンドラを記述する必要がある．

図 9 では，二分木型の接続網を宣言している．この例では，Anode エージェント型の宣言中で同じエージェント型のメンバ変数を定義している．この結果，**図 10** のように Anode 型のエージェントが再帰的に接続された二分木ネットワークが形成される．Orgel の接続宣言は静的であるので，論理的なモデルとしては

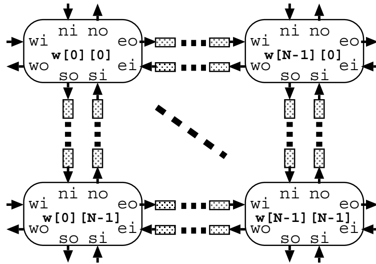

図 8 メッシュ型のプロセスネットワーク
Fig. 8 Mesh-type process network.

```
agent Anode(Sarc s: in){
    Anode nl, nr;
    Sarc sl, sr;
    connect self ==> sl ==> nl.s;
    connect self ==> sr ==> nr.s;
}
```

図 9 ツリー型結合網の接続宣言
Fig. 9 Connection declaration of tree network.

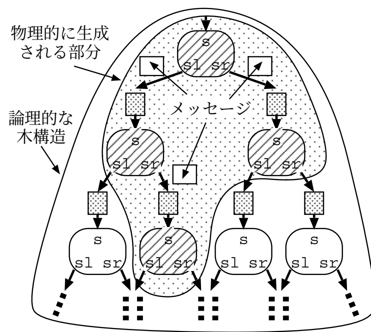

図 10 ツリー型のプロセスネットワーク
Fig. 10 Tree-type process network.

すべてのエージェントが 2 つの子を持つ無限に深い木構造となる．しかし実行時には，エージェントはメッセージ送信により遅延生成されるので，物理的に生成されるエージェントは親からのメッセージを受け取ったものだけになる．よって，子に渡す仕事がある場合のみメッセージを送信することで，必要なエージェントだけを物理生成するプログラムを記述できる．

このように，本記法は典型的な接続形態を簡潔に記述できる．一方で，部分的に接続形態が異なる場合には，表現が複雑になる．将来的には，必要に応じて疑似変数の値の範囲を明示的に指定できるようにすることを考えている．

2.7 メッセージの送受信

エージェント型のハンドラコード内でメッセージを送信するには，以下の形式の送信文を記述する．

Stream `<==` *Message*;

ここで，*Stream*には出力ストリームもしくは送信端への接続を宣言したストリーム型メンバ変数を指定し，*Message*には *Stream*のストリーム型宣言で列挙したメッセージ型のメッセージオブジェクトを記述する．

メッセージオブジェクトはメッセージ型名の後に，実引数として宣言と同じ型の値や変数を列挙する．引数の型は，ポインタを除く C のすべてのデータ型[☆]，あるいはメッセージ型である．前者の場合は実引数の値が，後者の場合はそのメッセージ型の論理ポインタが，それぞれメッセージオブジェクト内に格納される[☆☆]．メッセージの受信側では，このオブジェクトの引数値が，メッセージハンドラに記述した対応する引数変数に格納される．このため，PVM などの通信ライブラリのようにデータのパック/アンパックを意識することなく，簡潔に送受信を記述できる．なお，メッセージハンドラでのメッセージ引数は，同名のメンバ変数があればそれが値格納対象となり，ない場合はハンドラ内で同名の局所変数が暗黙に宣言されたと見なす．

Orgel のメッセージ型変数は一種の論理変数として振る舞い，メッセージオブジェクトの代入により一度だけ具体化される．この性質を用いると，メッセージ型引数を使ってデータの一部を後から送ったり，受信側が送信側に返信したりできる．

メッセージオブジェクトのメッセージ型引数に未具体化変数 m_s を指定し，受信側でこの引数を受ける変数が m_r だったとする．引数のモードが `in` なら m_r は m_s に束縛され，後に m_s にオブジェクトが代入されると，m_r も同じオブジェクトを指すようになる．同様にモードが `out` なら m_s が m_r に束縛され，受信側が m_r に代入したオブジェクトを送信側が m_s で参照できるようになる．

束縛されたメッセージ型変数が指すオブジェクトの引数を参照するには，以下の形式のメッセージ参照式を用いる．

Variable `?==` *MessageType* `[(`*Arg1* `[, ...])]`;

この参照式は，メッセージ型変数 *Variable*が未具体化なら具体化されるまで中断し，その後，`dispatch` と同様に変数 *Arg1, ...*を対応する引数に関連づける．

例として，図 2 のプログラムでの処理を考える．`main` 型エージェントからの `Mdata` 型メッセージを受け取ると，`Aworker` 型エージェントはデータの値をメンバ配列変数 `d` に格納し，`taskstate` の値を変更して `task`

[☆] ポインタを含まなければ，配列や構造体であってもよい．
[☆☆] 配列変数の場合は，変数名を記述することで配列全体が格納される．

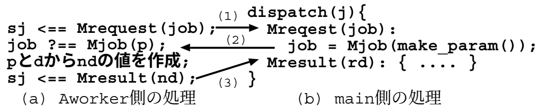

図 11　メッセージの送受信
Fig. 11　Message communication.

ハンドラを実行できるようにする．

同ハンドラでは，**図 11** (a) のコードを実行する．まず，sj に対して Mrequest 型メッセージを送る（図 11 (1)）．このメッセージの引数 job は仕事を受け取るための out モード引数なので，メッセージ参照式によりこれが具体化されるのを待つ．

一方，main 型エージェントでは図 11 (b) の dispatch 宣言により各ワーカーからのメッセージを処理する．Mrequest 型メッセージを受け取ると，メンバ関数 make_param() により仕事を生成してメッセージ引数を具体化する．この結果，暗黙の返信が行われて Aworker 側での参照式が成功する（図 11 (2)）．

Mjob 型メッセージによる返信を受け取ると，Aworker 側では与えられたパラメータに従った処理を行い，結果を Mresult 型メッセージで main に送る（図 11 (3)）．

Orgel ではメッセージ構造を静的に宣言するため，可変長の単一メッセージは作成できない．ただし，メッセージ型引数によって必要なだけメッセージをネストできるため，リストやツリーなどの非定型データの送信は容易に記述できる．しかし文字列のように単純な可変長バイト列を送る場合には，宣言したメッセージの大きさにデータを分割する必要がある．このため，将来的にはメッセージ引数として可変長配列を許すような拡張を考えている．

3．記述性と実行効率

2 章で述べたように，Orgel は抽象化された通信路を持つプロセスネットワークモデルに基づいている．また，エージェントやストリームは自動生成され，静的な接続宣言に従って自動的に接続される．以下，C と Pthreads を併用する場合，および，KL1 を用いる場合と比較し，こうした特徴の記述性と実行効率について考察する．プログラム例としては，図 1 にあげたマスターワーカー型の並列プロセスの記述を考える．

これの Orgel 版プログラムは，2 章で説明に用いた図 2 である．このプログラムではプロセスとその間の通信路をそのままエージェント型・ストリーム型として宣言し，ネットワーク構造は main 内で接続宣言により静的に記述している．各ワーカーからマスターへの仕事要求に対する返信処理も，メッセージの out モード引数を使うことで簡潔になっている．また，エージェント停止用のメッセージを追加する，ワーカーの下請エージェントを追加するといった，プログラムの拡張も容易である．

```
char data[M], rdata[N][M];
int param[N];
sem_t ww[N], wm;
pthread_mutex_t req_lock;

main(){
    int i, n;
    for (i = 0 ; i < N ; i++)
        pthread_create(..., worker, ...);
    while(...){
        sem_wait(&wm);
        pthread_mutex_lock(&req_lock);
        n = get_req();
        pthread_mutex_unlock(&req_lock);
        if (n & RESULT_MASK){
            n -= RESULT_MASK;
            if (rdata[n] が data より良い){
                data_writer_lock()
                data を更新;
                data_writer_unlock()
            }
        }
        else{
            param[n] = make_param();
            sem_post(&ww[n]);
        }
}
void worker(int *no){
    while (...){
        pthread_mutex_lock(&req_lock);
        put_req(*no);
        pthread_mutex_unlock(&req_lock);
        sem_post(&wm);
        sem_wait(&ww[*no]);
        data_reader_lock();
        data と param[*no] より rdata[*no] を更新;
        data_reader_unlock();
        put_req(*no + RESULT_MASK);
    }
}
```

図 12　マスターワーカー型のプログラム（Pthreads 版）
Fig. 12　Master-worker program (Pthreads version).

3.1　共有変数とプロセスネットワーク

同じ問題を C+Pthreads で記述したプログラムを**図 12** に示す．ただし，セマフォやロックなどの初期化処理は省略してある．また，put_req()，get_req() は整数を要素とするキューに値を出し入れする関数であり☆，data_reader_lock()，data_writer_lock()

☆ 排他制御部分を分かりやすくするため，ロックはこの関数の外で掛けている．

は配列 data を排他制御するためのリーダライタロック関数とする．

Pthreads を使う場合は共有メモリを前提としたプログラミングになるため，共有変数を用いた通信モデルを用いる．この場合，従来の逐次プログラミングに近い形で記述できるという利点があり，メッセージ構造の組立てや通信内容のコピーが必要ないため実行効率も良い．しかし，共有変数の排他制御を明示的に行う必要があるため，プログラマの負担は大きい．とくに，待ち時間やオーバヘッドを減らすには各変数の参照パターンを考慮してそれぞれ異なる制御方法をとる必要があり，コードの複雑さが増大する．

図 12 の例では data，rdata，param を共有変数とし，さらに rdata，param は配列化して各 worker ごとの領域を用意している．排他制御のコストを小さくするために，参照時間が長く書き込み頻度の低い data はリーダライタロックを用い，依存関係より main 側の参照前に worker 側の再書き込みが起こらない rdata はロックを省略するなど，個々の共有変数の性質を利用している．また，仕事要求と処理結果の通知はキューに貯めている．

実行効率の面では，主要なデータ構造を共有変数としたことで，値コピーのオーバヘッドが生じない分だけ Orgel より良い．しかし記述性の面では，本来のアルゴリズム中に低レベルな排他制御が混在するため，プログラムの見通しが悪い．また，上記のように効率化のため細かい工夫を行う場合，実装自体はさほど手間ではないものの，どの方式が効率的かはオーバヘッドと待ち時間のトレードオフになるため，判断が難しい．さらに，新たな共有変数を追加すると全体のアクセスタイミングが変わってくるため，機能拡張を行う場合には既存の共有変数も含めた見直しが必要になる．

Ada や Java のように，言語上でマルチスレッドをサポートする言語もいくつか実現されている[13]．しかしこうした言語では，Pthreads ライブラリを直接使う場合に比べてスレッドの生成・同期を比較的安全に行えるものの，この種の効率化の難しさについては解決されていない．

これに対してプロセスネットワークのようなメッセージパッシングモデルは，多少のオーバヘッドをともなうものの，単純なコーディングで安定した性能を発揮できる．さらに Orgel では，ネットワーク接続が静的であることから，書き込み/読み出し側が単数/複数の場合にそれぞれ異なる排他制御コードを生成するなど，処理系で最適化を行いやすい．

なお，Pthreads と同様な低レベルの並列化ライブラリである PVM では，メッセージパッシング型のプログラミングモデルとなる．このため Pthreads のように複雑な同期・排他制御を使い分ける必要はなく，共有メモリ/分散メモリ環境のいずれでも同一のプログラムを動かすことができる．しかし，Orgel のように自動的なメッセージディスパッチ機構は用意されておらず，非同期通信の記述に手間が掛かる．また，メッセージ構造が抽象化されていないため，その要素となる個々の C のデータ型単位でパック/アンパックを記述しなければならない．

```
main
 :- generic:new(merge,{SjI}, SjO),
    D = data(...),
    create_worker(8, Sc, SjI, D),
    master(SjO, Sc, D).
create_worker(N, Sc, SjI, D) :- N > 0
  | worker(Sc, SjI0, D), SjI = {SjI0, SjI1},
    N0 := N-1, create_worker(N0, Sc, SjI1, D).
create_worker(N, Sc, SjI, D) :- N =:= 0
  | SjI = {}.

master([req(J)|Sj], Sc, D)
 :- make_param(..., J), master(Sj, Sc, D).
master([result(RD)|Sj], Sc, D)
 :- update_data(RD, D, D0, Sc, Sc0),
    master(Sj, Sc0, D0).
update_data(RD, D, D0, Sc, Sc0) :- RDがDより良い
  | D0 = RD, Sc = [data(D0) | Sc0].
otherwise.
update_data(RD, D, D0, Sc, Sc0)
 :- D0 = D, Sc = Sc0.

worker([data(D0)|Sc], Sj, D)
 :- worker(Sc, Sj, D0).
alternatively.
worker(Sc, Sj, D)
 :- Sj = [req(J)|Sj0], worker1(J, Sc, Sj0, D).
worker1(job(Param), Sc, Sj, D)
 :- ParamとDからD0を作成,
    Sj = [result(D0)|Sj0], worker(Sc, Sj0, D).
```

図 13 マスターワーカー型のプログラム（KL1 版）
Fig. 13 Master-worker program (KL1 version).

3.2 静的/動的なプロセスネットワーク

KL1 で記述したプログラムを**図 13** に示す．ただし，ファンクタ data の詳細は省略してある．

KL1 ではプロセスやストリームは文法的に定義されているわけではなく，ゴールの再帰呼出により持続的に存在するプロセスを作り，リスト構造を部分的に具体化していくことでストリーム通信を行う．

たとえば図 13 の場合，main/0☆節から呼び出され

☆ 論理型言語の述語は '述語名/引数の個数' という形式で記述する．

た master/3 と，ゴール create_worker/4 により呼び出された N（図では 8）個のゴール worker/3 は，それぞれ再帰呼出によりプロセスとして存在し続ける．

master/3，worker/3 は引数として共有変数を渡されており，これらを用いてストリーム通信を行う．master/3 から worker/3 への通信はマルチキャストなので，たんに共有変数を各 worker/3 に渡している．一方，逆方向の通信は複数の worker/3 より届くメッセージ列のマージなので，組込オブジェクトとして用意されたマージャを生成して，各 worker/3 からの出力ストリームをマージしている．この結果，Orgel と同様に図 1 のプロセスネットワークが構築される．

このようなプロセスネットワークの構築方法は，文法が非常に単純で覚えやすいという論理型言語の利点を生かしている．その反面，プログラム中のどの部分がプロセスやストリームになるのかを指示する構文がないため，ユーザにとってネットワークモデルが明確ではなく，新たなストリームを追加するといった拡張を行う際には誤りを生じやすい．さらに，宣言的な言語であるにもかかわらずプロセスネットワークの構造は静的に定義されず，むしろ操作的に作られる．したがって，存在しないプロセスにメッセージを送るといった同期タイミングのバグは防止できるものの，意図しない構造のネットワークを構築してしまうという誤りは生じうる．

動的な処理の記述能力については，KL1 は非常に強力である．メッセージには任意の KL1 データを含めることができ，未具体化引数による細かい同期制御を行ったり，述語やストリーム変数を送信したりできる．

これに対し Orgel では，メッセージ型やネットワーク接続が静的であるという制約が加えられている．

前者については，動的に新たなメッセージ構造を作れないものの，メッセージ型引数のネストによって，リストや木といった動的・非定型な構造のメッセージを作成できる．アプリケーション記述においてメタなメッセージ生成を必要とすることは少ないから，この制約はほとんど問題を生じず，メッセージの型チェックが静的にできることや粒度解析を行いやすいことなどの利点の方が大きい．

一方，後者についてはプログラムの記述力に明らかな制約が生じる．動的なネットワークの生成・削除ができないため，必要に応じてネットワークを拡大・縮小するようなプログラムは記述できないし，動的に接続相手を変更することもできない．このため，OS などのシステム記述は非常に困難であり，アプリケーション記述でもこのようなプログラムを必要とする分野には適していない．

しかし非定型的な非数値プログラムでも，回路シミュレーションや遺伝的アルゴリズム処理など静的なプロセスネットワーク上で非均質な通信を行う場合には，非同期的なメッセージ送受信や非定型データを扱う能力があればよい．さらに，非数値分野の主要な問題の 1 つである木探索についても，2.6 節の図 9，10 の例に示したように，再帰的なプロセスネットワークの定義によって扱うことができる．

また，現在はエージェント型/ストリーム型の配列変数は静的に大きさが決まるため，問題の大きさに応じてプロセスネットワークの大きさが変わるような記述はできない．このため，そのつどプログラムを再コンパイルするか，予想される最大の大きさでネットワークを定義しておいてその一部を使うという形で対応するしかない．

この問題については，単一代入の共有変数を導入し，この変数による配列要素数の指定を許すという拡張を考えている．プロセスネットワーク中で問題の大きさに依存する部分がメッセージ受信により生成されるのは，大きさが決定した後と考えられる．したがって，問題の大きさが決定した時点でこの変数に値を代入するようにプログラムを記述すれば，現在の静的なネットワーク記述の枠組みを崩さずに実行時にネットワークの大きさを決定できる．

このように，非数値型アプリケーションでも Orgel のプロセスネットワーク記述力で十分な場合は多く，いくつかの問題点に対する言語拡張を施すことで，高い実用性が得られる．このため，動的操作の禁止など記述力を制限することで得られる，バグ防止や最適化効果などの利点の方が大きいと考える．

4. 実 装

実装した Orgel 処理系は，エージェントの並行/並列実行に Pthreads ライブラリを利用し，スレッド間で共有するメモリ空間を用いて通信を行う．そのため，逐次環境での並行実行，ならびに，メモリ共有型並列環境での並列実行を行うことができる．

4.1 処理系の構成

共有メモリ版処理系は，Orgel コンパイラとランタイムライブラリで構成される．

Orgel コンパイラ Orc は，Orgel から C へのトランスレータとして実装した．この出力を既存の C コンパイラに渡し，Orgel ランタイムライブラリとリンクすることで，実行コードを生成する．

Orc はユーザが記述した Orgel プログラムについ

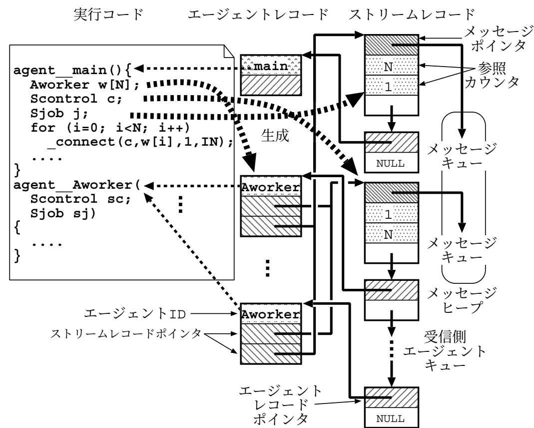

図 14 ランタイムの構造
Fig. 14 Structure of the Orgel runtime.

て，メッセージやストリームなど Orgel 固有の宣言文は，C の定数や構造体の定義に変換する．エージェント型宣言のハンドラ部は逐次の C コードなので，基本的にそのまま出力し，メッセージ送信文など Orgel 独自の構文のみ，その機能を実現する C コードやランタイム関数の呼び出しに置き換える．

このとき Orgel の言語上の性質を利用してプログラム中の可能な要因はコンパイル時に解決し，できる限り静的なコード生成を行う．たとえば，ストリーム型が受け付けるメッセージ型やメッセージ引数の型は宣言されているため，これに反するコードはコンパイル時にエラーにでき，誤った型の受信や格納に対する実行時エラー処理は必要ない．また，メッセージ型引数の具体化/参照の流れはモードより静的に決定するから，メッセージ型変数間の束縛コードを単純化できる．

4.2 ランタイムの構造

実装したランタイムでは，エージェントやストリームのインスタンスをそれぞれエージェントレコード，ストリームレコードで表す（**図 14**）．

同じストリームを通るメッセージは送信順の受信が保証されるから，ストリームはメッセージキューとして実現できる．しかし，複数のストリームの間ではメッセージの処理順序が決まっていないうえ，メッセージ型により必要なメモリサイズも異なる．そこで，メッセージはガーベジコレクション（GC）機能を持つヒープ領域上に生成し，ストリームレコードからこのヒープ上のメッセージキューを指すようにしている．

4.3 エージェントとストリームの生成

エージェントは，それぞれ 1 スレッドとして生成することで，並行/並列実行を行う．

コンパイラは，図 3 に示した形式のエージェント型宣言それぞれについて，エージェントメイン関数を生成する．この関数は，`initial` ハンドラを呼び出してからメインループに入り，メッセージが到着していれば対応するハンドラを，到着していない場合は `task` ハンドラを呼び出す．`terminate` 文はこのメインループを終わらせるコードに置換し，ループを抜けると `final` ハンドラを呼び出してから終了する．

単純に考えると，定義された個々のエージェント型変数に対してスレッドを生成し，変数の型に対応するエージェントメイン関数を実行させればよい．しかし 2.5 節で述べたように，最初のメッセージ受信時までエージェントの生成を遅延する必要がある．そこで，あるエージェントメイン関数が実行されたとき，その中で定義されたエージェント/ストリーム型メンバ変数についてはレコードの生成のみ行い，以下の手順で接続情報を格納しておく（図 14）．

(1) エージェントレコードに，エージェント型を表す ID を格納する．また，同レコードの引数領域に，接続宣言の指定に対応するストリームレコードへのポインタを格納する．

(2) ストリームレコードに，接続された送信/受信エージェント数（参照カウンタ）を格納する．

(3) 受信側に接続されたエージェントレコードでキューを形成し，その先頭へのポインタをストリームレコードに格納する．

後にあるストリームにメッセージが送信されたとき，(3) のキューをたどって受信側エージェントのスレッドを生成し，キューを破棄する．各スレッドは自分のエージェントレコードを利用し，(1) の ID よりテーブルを引いて対応するエージェントメイン関数のポインタを取得し，実行を開始する．また，レコードの引数領域より入出力ストリームへのポインタを取得する．なお，あるエージェントの入力ストリームが複数ある場合，どのストリームにメッセージが送られてもこのエージェントをスレッド化する必要がある．このため受信側エージェントキューは間接参照構造になっている．

一方，これらのレコードは不要になった段階で解放し，再利用する必要がある．そこでエージェントメイン関数の最後で自身のエージェントレコードを解放し，同時に (2) で格納しておいたストリームレコードの参照カウンタから自分の分を減らす．この結果，カウンタが 0 になれば，そのストリームに送受信するエージェントはいなくなったのでストリームレコードを解放する．

共有メモリ環境下では生成したスレッドに OS が自

動的に CPU を割り当てるため，現在の実装ではエージェントのスケジューリングは OS に任せている．将来のバージョンではスレッドの優先度を利用し，さらに sched_yield() を呼び出して CPU を放棄するコードを埋め込むことで，スケジューリングの最適化を実現する予定である．

4.4 メッセージヒープ

4.2 節で述べたように，メッセージオブジェクトはヒープ上に生成し，GC によりメモリの再利用を行う．

KL1 処理系 KLIC [14]でも，本処理系と同様に C へのトランスレータ方式を採用し，GC 機能を持つヒープによるデータ領域管理をしている．しかし，KLIC ではすべてのデータがゴールレコードから指されるため，ゴールの集合が GC ルートの役割を果たすのに対し，本処理系のメッセージオブジェクトは各スレッド上の C レベルの変数から指されている．このため，生き残っているオブジェクトの集合を求めたり，オブジェクトの移動に合わせて，メッセージ型変数が移動先を指すように書き換えたりすることが難しい．

そこで**図 15** のように，メッセージヒープをエントリテーブルとヒープ領域の 2 段構成で実装し，参照カウンタ/コピー方式を組み合わせた GC を行っている．ヒープ領域に後者の GC を用いるのは，Orgel ではヒープに置かれるのはメッセージオブジェクトだけであることから，そのほとんどが短期間で不要になりコピー対象が少ないことが期待できるからである．

ヒープ領域はメッセージオブジェクト格納用のメモリ領域である．Orgel の各メッセージ型は固定長なので，コンパイル時にそれぞれ C の構造体型に変換しておく．実行時には，ヒープ領域の空き領域から必要な大きさを割り当て，この構造体型にキャストすることで，各引数領域をメンバとしてアクセスする．

エントリテーブルには，各メッセージオブジェクトの参照カウンタとアドレスを格納している．実行時にオブジェクトを生成するときは，ヒープ領域を割り当てると同時にエントリを 1 つ確保し，参照カウンタを 1 にして割り当てた領域のアドレスを格納する．以下，このエントリをヒープエントリと呼ぶ．確保したヒープエントリは，アクティブエントリキューにつなぐ．

Orc は Orgel プログラム中のメッセージ型変数を，すべてメッセージポインタ型変数に置き換える．また，メッセージ型を表す構造体についても，メッセージ型引数はメッセージポインタ型のメンバに変換しておく．以下，これらをメッセージポインタと呼ぶ．メッセージポインタの値はヒープ領域中のアドレスではなく，対応するヒープエントリのインデックスを表す整数値

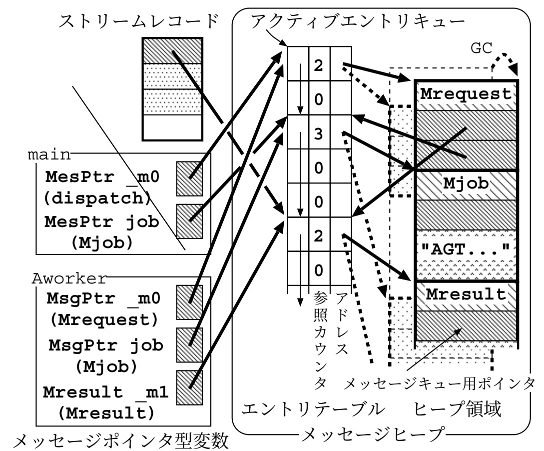

図 15 メッセージヒープの実装
Fig. 15 Implementation of the message heap.

となっている．メッセージオブジェクトを操作する際には，まずエントリテーブルを引いてメッセージポインタの値をオブジェクトのアドレスに変換する．

メッセージの送受信などでメッセージポインタの値をコピーするときは，対応するヒープエントリについて参照カウンタを増やす．また，メッセージポインタ型変数を定義したスコープ外に実行が進むときやメッセージ型引数を含むオブジェクトを破棄するときは，参照カウンタを減らす．このようなカウンタ操作コードをコンパイラが埋め込むことで，参照カウンタの値はそのオブジェクトを指すメッセージポインタの個数に保たれる．

参照カウンタが 0 になると，ヒープエントリを回収し，新たなオブジェクトを作るときに再利用する．一方，到達不能になったヒープ領域上のオブジェクトはそのまま放置する．やがてヒープ領域がいっぱいになったとき，アクティブエントリキューをたどり，まだ参照されているオブジェクトだけを同サイズのヒープ領域へとコピーする．このとき，ヒープエントリが持つオブジェクトのアドレスをコピー先のアドレスに書き換えることで，各メッセージポインタの内容を書き換えることなくオブジェクトを移動できる．

4.5 メッセージの送受信

送信文によるメッセージ送信とメッセージハンドラによる受信は，排他制御を行ってストリームレコードにアクセスし，メッセージキューを操作する．各ストリームレコードは条件変数を持ち，新着メッセージがない場合に受信スレッドはこれを使って中断する．そして送信スレッドがシグナルを送ることで，中断スレッドを再開させる．

メッセージ型引数によるメッセージ型変数の束縛に

ついては，以下のように処理する．すでに具体化済なら参照側が具体値オブジェクトのヒープエントリを指すようにする．未具体化であれば，まだオブジェクトを指していないヒープエントリを新たに割り当て，双方のメッセージポインタがこのエントリを指すようにする．このヒープエントリは，後に具体化が起きた時点で具体値オブジェクトを指すように書き換えられ，同時に参照側も同じ値を参照できるようになる．Orgel では引数のモードによりデータフローの方向が決まっているから，親メッセージの送受信者が同時に具体化してしまった場合の競合解決を考える必要はない．

また，具体化の前にもう一方が参照してしまった場合，参照側は条件変数を持つフックオブジェクトをヒープエントリにつなぎ中断する．具体化側は，具体化を行う際にヒープエントリがフックオブジェクトを指していれば，シグナルを送って参照側を再開させる．

5. 性 能 評 価

以下の問題を用いて，実装した処理系を評価した．

nqueen N-queen 問題を解くベンチマークプログラムであり，$N = 13$ とした．

pia 並列反復改善法によるタンパク質配列のマルチプルアライメントを行う応用プログラムであり[15]，長さ 80 の配列 15 本のマッチングを行った．

これらの問題を同じアルゴリズムで解くプログラムを以下の言語・ライブラリで作成し，比較を行った．

(1) C（逐次）
(2) C+Pthreads ライブラリ
(3) Orgel
(4) KL1（KLIC 2.002 版☆）

5.1 nqueen

各言語で記述したプログラムの大きさを，**表 1** に示す．ソース行数は全ソースからコメント行と空行を除いた行数であり，括弧内は並列化のために C 版に追加/変更したコードの行数である．バイナリサイズは実行ファイルのバイト数である．

また，SPARCcenter（20 プロセッサ，OS: Solaris 2.6）上での実行時間を**表 2** に示す．速度向上率は各並列プログラムの C 版に対する実行時間の逆比であり，括弧内は同じプログラムの逐次実行時間に対する逆比（台数効果）である．また，オーバヘッドは逐次環境での C 版に対する実行時間増加分の比（%）である．

このプログラムは，第 1 列の各行に最初のクイーンを置いた場合それぞれを初期位置とし，各初期位置について残りの列の可能な配置を全解探索する．得られた解は最終的に 1 個のプロセッサに集めているため，並列実行の際には探索/収集を合わせて 14 プロセッサを使用する．

Pthreads と Orgel のプログラムは，C を並列化する形で作成した．Pthreads 版では探索関数を単純にスレッド化したのに対し，Orgel 版はエージェントやストリームの宣言を追加している．しかし，前者が解の収集時に共有変数の排他制御を行う必要があるのに対し，後者はメッセージ送信文 1 行で済んでいる．このため，変更した行数は同程度であり，実際の変更の難易度は Orgel の方が低い．実行効率はともに同程度であり，10〜11 倍と高い速度向上を得ている．また，逐次実行のオーバヘッドもわずか 3%程度であった．

KL1 の場合は，抽象度の高い記述が可能であるため，並列プログラムでありながらソース行数は逐次の C 以下である．このため，C のコードを流用できないとはいえ生産性は高い．しかしその代償として組込述語や汎用単一化などの大規模なランタイムを必要とし，バイナリサイズは C 版の 50 倍にもなる．また，台数効果は Pthreads 版や Orgel 版を上回るものの，ランタイムのオーバヘッドにより，他言語版の 5 倍ほど遅く，速度向上率は 2 倍程度しか得られていない．

Orgel の場合も，メッセージ型変数の具体化やヒープによるメモリ管理の機能を必要とするが，型やメッセージの方向が静的に決まるため，多くの要因がコンパイル時に解決する．このため，ランタイムは小規模なものですみ，バイナリサイズは Pthreads 版の 6 割増し程度に収まっている．また，上で述べたとおり，実行時オーバヘッドは非常に小さい．

表 1 nqueen のプログラムサイズ
Table 1 Program size of nquen.

	ソース行数（変更行数）	バイナリサイズ
C	51 (–)	23765
Pthreads	106 (66)	30484
KL1	46 (46)	1042808
Orgel	107 (71)	50385

表 2 nqueen の実行速度
Table 2 Execution speed of nqueen.

	逐次 (s)	並列 (s)	速度向上率	overhead
C	28.78	–	–	–
Pthr.	29.67	2.56	11.24 (11.59)	3.1
KL1	151.94	12.68	2.27 (11.98)	427.9
Orgel	29.74	2.79	10.32 (10.66)	3.3

☆ 2001 年 5 月現在，3.003 版まで出ているが，共有メモリ環境版にはバグがあるため，2.002 版を使用した．

表 3 piaのプログラムサイズ
Table 3 Program size of pia.

	ソース行数（変更行数）	バイナリサイズ
C	5218 (–)	106196
Pthreads	5291 (95)	112946
KL1	5444 (1179)	1223648
Orgel	5332 (183)	139427

表 4 piaの実行速度
Table 4 Execution speed of pia.

	逐次 (s)	並列 (s)	速度向上率	overhead
C	76.72	–	–	–
Pthr.	79.99	10.95	7.01 (7.31)	4.3
KL1	91.82	12.52	6.13 (7.33)	19.7
Orgel	80.56	11.06	6.94 (7.28)	5.0

5.2 pia

nqueenと同様に，各言語で記述したプログラムの大きさと実行速度を，それぞれ**表 3**，**表 4** に示す．

このプログラムは，文字列で表現した N 本のタンパク質配列に対し，可能な N 通りの組合せについて 1 対 $(N-1)$ 本のマッチングを行う．得られた N 通りの結果からコスト最小のものを選び，収束するまで同様なマッチングを繰り返す．並列モデルとしては，N 個の並列実行可能なマッチング部と最小コスト計算部とが，双方向通信をしながら交互に動作する形になる．

解が単方向に収集されるだけのnqueenと比較すると，piaでは文字列の配列およびコストなどのパラメータという複雑なデータが，マッチング部と最小コスト計算部との間で双方向にやりとりされる．

このプログラムはKL1版がオリジナルであり，文字列操作のオーバヘッドを減らすため，マッチングを行うコードはC，全体を並列制御する部分はKL1という組合せで記述されている．したがって，他言語版はKL1部分を書き直す形で移植した．

Pthreads版は文字列配列などを共有変数とし，マッチング部と最小コスト計算部が交互にこれを読み書きする．したがって，通信オーバヘッドは最小で済んでいる．その一方で，単なる排他制御ではなく 1 対多/多対 1 の待合せが必要なため，ロックと条件変数を用いた複雑な同期コードを記述する必要があった．

Orgelでは，これらのデータのやりとりをすべてメッセージ通信で行う．このため，データの待合せは自動的に行われ，1 対多/多対 1 通信も接続宣言 1 行で記述できる．よって，並列モデルに従った素直なコード記述が可能であり，Pthreads版よりも変更行数は多いにもかかわらず，その労力は圧倒的に少ない．メッセージ生成のたびにデータのコピーが生じるが，そのオーバヘッドはPthreads版と比較して許容範囲と思われる．

上記のようにKL1版は計算の核となる部分がCで記述されているため，実行効率は他言語版に匹敵する．しかし，nqueenのようにソースが短くなるという利点は得られていない．また，バイナリサイズも他言語版の 10 倍ほどになっている．

6. 関連研究

並列言語については，従来より多くの研究が行われている[16]．

並列プロセス間の通信路接続を静的に行う言語は，Occam[17]をはじめとして，数多く提案されている．しかしこれらの多くは定型的な並列記述に適しており，我々が目的とする非定型的な非数値処理には不十分な点が多い．一方でKL1[4),18]やABCL/1[5]のように，柔軟な並列プロセス記述と動的データに適した抽象度の高い通信機構を提供する言語の研究も行われている．だが，これらの多くではプロセスネットワークやメッセージ構造が動的に決定するため，効率的な実装が難しい．

Orgelでは，非数値アプリケーション記述に必要な範囲で後者の強力な記述能力を残しつつ，ネットワークやメッセージに宣言的な記法を取り入れることで，前者のような効率的な実装を可能にすることを狙っている．

6.1 プロセスネットワークの静的記述

OccamではPAR構文により並列プロセス構造を定義し，それらの間をチャネルで接続する．これにより，静的なプロセスネットワークを記述することができる．しかしチャネルには，接続が 1 対 1 のみである，データがバッファリングされないため受信側と同期するまで送信側が待たされる，チャネル宣言時に指定した単一のデータ型しか送信できない，など制限が強い．

より強力な記述モデルはいくつか提案されており，たとえばCommunication skeletons[19]ではプログラムを計算ステップと通信ステップに分け，各通信ステップに対して抽象化された通信パターンを記述する．だが我々が想定している分野では，粒度の異なる計算要素間で非同期的な通信が発生するから，このように定型的なステップ構造では表現できない．

Orgelに近いモデルや言語としては，Candidate Type Architecture (CTA)[20]やNCC[21]，Darwin[22]などがあげられる．

CTAではプログラムを，個々の逐次処理を記述す

るXレベル，Xレベル処理の結合を記述するYレベル，全体の制御を記述するZレベル，という3階層で記述する．プロセスネットワークはYレベルでポートの接続を静的に定義することで実現するため，各レベルがそれぞれOrgelのエージェント関数，接続宣言，エージェント型宣言に対応する．

NCCはCの拡張になっており，プロセスに入出力ポートのインタフェースを記述することや非決定的な受信機構を持つことなど，Orgelとの共通点も多い．通信路はマルチキャストが可能であるが，メッセージキューではなく受信側にサイズ1のバッファしか持たない．

Darwinではコンポーネントを定義し，それらをポータルで接続することでプロセスネットワークを構築する．宣言したコンポーネント型の変数を定義することで，実行時にコンポーネントが自動的に生成されること，`bind`宣言文によりポータルの静的な接続を行うことなど，Orgelによく似た言語仕様になっている．また，Orgelにない特徴として，コンポーネント外に対してポータルが可視/不可視の属性を持つなど，コンポーネントをカプセル化するための機構を持つ．

これらのモデルや言語ではOrgelと同様に，複雑なプロセスネットワークモデルを記述できる．しかしポートやポータルといったプロセスの通信口どうしを直結しているため，通信網の制約が大きい．たとえばDarwinのポータルは1対多の接続をサポートしているが，多対多の接続はできない．また，インタフェース宣言によって複雑なデータ型のやりとりを可能にしているが，Orgelのネストしたメッセージのような動的な構造を送ることはできない．これに対して，Orgelではストリームを介することで，通信網の記述能力をより強力にしている．

6.2 非同期受信機構

個々の並列実行単位が受信機構を持ち，非同期に到着するメッセージに反応するという機構は，actorモデル[23]に基づいている．しかし，同じモデルによるABCL/1やKL1などが細粒度の並列性を抽出しようとするのに対し，Orgelでは逐次性の高い部分をエージェント関数として記述することで，並列実行単位であるエージェントの粒度がある程度の大きさになることを期待している．このため，知識や信念に対する言語サポートはないものの，並列モデルとしてはShohamのAOP（agent-oriented programming）[9]に近い．

6.3 ストリーム通信

未具体化リストを使って連続的なメッセージの受け渡しを行うストリーム通信の手法は，KL1やPCN[24]などで使われている．これらの言語では，ストリームはあくまで一部が未具体化のリストデータであり，共有データにアクセスした際のランタイムの振舞いを利用して，通信や同期を実現している．このようにストリームとその上を流れるデータが一体化しているため，ストリーム自体を他の並列単位に引き渡すなど柔軟な操作が可能である反面，通信パターンを静的に解析することが難しい．またKL1の場合，再帰的なゴール呼出による並行プロセスの実現手法も，プロセスの分割や移動が自由に行える一方で，プロセスネットワークの構造の静的解析を困難にしている．こうした強力な言語機能はOS記述[25]などで役立つ反面，一般のアプリケーション記述では過度な部分もある．

そこでOrgelでは，ストリームで接続されたエージェントというプロセスネットワークの型をはめることで，必要な記述性を保ったまま効率的な実装を可能にすることを狙っている．KL1に同様な制約を加えてアプリケーションを記述しやすくした既存言語には，並列オブジェクト指向言語A'UM[12]やプロセス指向言語AYA[26]がある．これらの言語ではオブジェクトやプロセスといった並列実行単位を文法レベルで定義し，それらの間の通信路としてストリームを位置づけている．Orgelではこうした制約に加え，ストリームの接続やその上を流れるメッセージを静的に定義するようにしている．

7. おわりに

本論文では，我々が開発している非数値向け並列言語Orgelと共有メモリ型並列計算機への実装について述べ，性能評価の結果を示した．

Orgelでは記述のしやすさと実行効率の両立を目指し，ストリームで接続されたエージェント群というプロセスネットワークを宣言的に記述する．プログラム記述時/コンパイル時に明確な並列モデルを得ることで見通しの良い並列プログラム記述を可能にすると同時に，コンパイル時に決定可能な要因を増やして最適化しやすくしている．その一方で，ストリームによる強力な通信機構を用いて簡潔なメッセージ通信が記述できる．

実装した処理系を性能評価した結果，直接Pthreadsライブラリを用いる場合と比較して，非常に見通しの良いプログラミングが可能であった．また，ランタイムによるオーバヘッドやバイナリサイズの増加は十分小さく，KL1のような動的要因の多い言語の処理系と比較して，非常に効率が良いことが示された．今後は，より非定型的なプログラムについても評価が必要

である．

実装済の逐次/共有メモリ版処理系に加え，現在は分散メモリ版処理系を実装中である．また，静的/動的処理を組み合わせたスケジューリング最適化手法の設計・実装を進めている．これらについても今後，評価を行っていく予定である．

謝辞 数々の有益なコメントをいただいた査読者の方々に感謝いたします．なお，本研究の一部は日本学術振興会「未来開拓学術研究推進事業」および（財）堀情報科学振興財団の助成による．

参 考 文 献

1) Sunderam, V.S.: PVM: A framework for parallel distributed computing, *Concurrency: Practice and Experience 2*, Vol.2, No.4, pp.315–339 (1990).
2) Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard* (June 1995).
3) Nichols, B., Buttlar, D. and Farrell, J.P.: *Pthreads Programming.* O'REILLY (1998).
4) Ueda, K. and Chikayama, T.: Design of the kernel language for the parallel inference machine. *The Computer Journal*, Vol.33, No.6, pp.494–500 (1990).
5) Yonezawa, A. (Ed.): *ABCL An Object-Oriented Concurrent System: Theory, Language, Programming, Implementation, and Application.* MIT Press (1990).
6) 大野和彦，岡野孝典，山本繁弘，中島 浩：マルチエージェントパラダイムと宣言的通信ストリームに基づく並列言語，情報処理学会第 27 回プログラミング研究会 (Jan. 2000).
7) Ohno, K., Yamamoto, S., Okano, T. and Nakashima, H.: Orgel: A parallel programming language with declarative communication streams, *Proc. 3rd International Symposium ISHPC 2000*, pp.344–354 (Oct. 2000).
8) 丸山真佐夫，山本繁弘，大野和彦，中島 浩：並列プログラムデバッギングのための巻き戻し実行機構，並列処理シンポジウム，pp.89–90 (June 2001).
9) Shoham, Y.: Agent-oriented programming, *Artificial Intelligence*, Vol.60, pp.51–92 (1993).
10) Hewitt, C.: Viewing control structures as patterns of passing messages, *Artificial Intelligence*, Vol.8, pp.323–364 (1977).
11) Agha, G.: Concurrent object-oriented programming, *Comm. ACM*, Vol.33, No.9, pp.125–141 (1990).
12) Yoshida, K. and Chikayama, T.: aum—a stream-based concurrent object-oriented language, *New Generation Computing*, Vol.7, No.2, pp.127–157 (1990).
13) Lewis, B. and Berg, D.J.: *Multithreaded Programming with Pthreads*, Prentice-Hall Inc. (1998).
14) Fujise, T., Chikayama, T., Rokusawa, K. and Nakase, A.: KLIC: A portable implementation of KL1, *Proc. International Symposium on FGCS'94*, pp.66–79 (Dec. 1994).
15) Ishikawa, M., Hoshida, M., Hirosawa, M., Toya, T., Onizuka, K. and Nitta, K.: Protein sequence analysis by parallel inference machine, *Proc. 5th Gener. Comp. Sys.*, pp.294–299 (1992).
16) Skillicorn, D.B. and Talia, D.: Models and languages for parallel computation, *ACM Computing Surveys*, Vol.30, No.2, pp.123–169 (1998).
17) SGS-Thomson Microelectronics: *occam 2.1 reference manual* (1995).
18) Ueda, K.: Guarded horn clauses, *LNCS221 Logic Programming '85*, pp.168–179, Springer-Verlag (1986).
19) Skillicorn, D.B.: Communication skeletons, *Abstract Machine Models for Parallel and Distributed Computing*, Kara, M., Dave, J.R., Goodeve, D. and Nash, J. (Eds.), pp.163–178, IOS Press (Apr. 1996).
20) Alverson, G.A., Griswold, W.G., Lin, C., Notkin, D. and Snyder, L.: Abstractions for portable, scalable parallel programming, *IEEE Trans. Parallel and Distributed Systems*, Vol.9, No.1, pp.71–86 (1998).
21) 朴 泰佑，工藤知宏，天野英晴，木村哲郎：マルチプロセッサのための科学技術計算用並行記述言語 NCC，電気情報通信学会論文誌，Vol.J72-D-I, No.10, pp.750–758 (1989).
22) Eisenbach, S. and Patterson, R.: Pi-calculus semantics of the concurrent configuration language Darwin, *Proc. 26th Annual Hawaii International Conference on System Science*, Vol.2 (1993).
23) Agha, G.: *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass. (1986).
24) Foster, I., Olson, R. and Tuecke, S.: Productive parallel programming: The PCN approach, *Scientific Programming*, Vol.1, No.1 (1992).
25) Chikayama, T.: Operating system PIMOS and kernel language KL1, *FGCS'92*, pp.73–88 (1992).
26) 寿崎かすみ（ICOT TM-1206）：aya 第 1 版解説書.

(平成 13 年 5 月 8 日受付)
(平成 13 年 9 月 14 日採録)

大野　和彦（正会員）

1998 年京都大学大学院工学研究科情報工学専攻博士後期課程修了．同年豊橋技術科学大学助手．並列プログラミング言語の設計と最適化に関する研究に従事．博士（工学）．

山本　繁弘

1977 年生．2000 年豊橋技術科学大学情報工学科卒業．現在同大学大学院修士課程情報工学専攻在籍．並列言語のコンパイラに関する研究に従事．

岡野　孝典

1978 年生．2000 年豊橋技術科学大学情報工学科卒業．同年（株）日立システムアンドサービス入社．在学中は並列言語の実装に関する研究に従事．

中島　　浩（正会員）

1981 年京都大学大学院工学研究科情報工学専攻修士課程修了．同年三菱電機（株）入社．推論マシンの研究開発に従事．1992 年京都大学工学部助教授．1997 年豊橋技術科学大学教授．並列計算機のアーキテクチャ等並列処理に関する研究に従事．工学博士．1988 年元岡賞，1993 年坂井記念特別賞受賞．IEEE-CS，ACM，ALP，TUG 各会員．