Sushiによるアノテーションプログラミング

菅野博靖　山中英樹　鵜飼孝典
{suga,yamanaka,ugai}@iias.flab.fujitsu.co.jp
(株) 富士通研究所　情報社会科学研究所
〒261 千葉市美浜区中瀬1−9−3

並列分散プログラミング言語 Sushi (Smartly User Schedulable HIgh-level Language) のアノテーション機構について報告する．Sushi は，分散メモリ型並列コンピュータやワークステーションクラスタ上において，動的な資源管理をユーザに扱い易く提供することを目的として設計された言語である．その本質は，ストリーム計算をベースにした計算プロセスの記述と動的な計算資源の管理を記述するアノテーションを分離することにより，問題固有のアルゴリズムの記述と計算機環境に依存した情報の記述を分離できることにある．本稿では，アノテーションによる動的資源管理や性能チューニング，さらに実行メカニズムについて報告し，考察を行う．

Annotation Programming in Sushi

Hiroyasu Sugano　Hideki Yamanaka　Takanori Ugai
{suga,yamanaka,ugai}@iias.flab.fujitsu.co.jp
Institute for Social Information Science,
FUJITSU LABORATORIES LTD.
1-9-3 Nakase, Mihama-ku, Chiba 261 Japan.

We developed a parallel/distributed programming language Sushi (Smartly User Schedulable HIgh-level Language) with its Annotation mechanism. A Sushi program consists of two parts: the Annotation part and the Computation part. The former is a machine- and environment-dependent specification for dynamic resource management, and in the latter, a usual domain specific algorithm is described. Dynamic resource management and performance tuning in the parallel/distributed computing environment can be realized flexibly by programming in the Annotation.

1 はじめに

分散メモリ型並列コンピュータやワークステーションを利用した分散計算機環境が身近になり，並列分散ソフトウェアの効率的な開発環境が求められている．これまでの並列分散ソフトウェア構築では，熟達したプログラマでさえ計算資源を効率良く利用するためには多くの工数を割かねばならなかった．分散計算機環境を有効に活用するための資源管理には一般的な方法論が存在せず，問題に応じた資源配分戦略をプログラム中に記述する必要があるためである．これは，並列分散プログラミングに不慣れなユーザには特に大きな壁となっていた．エキスパートのプログラマはもちろん，こうしたエンドユーザにも利用しやすいプログラミング環境を構築することは，当り前のものとなってきた並列分散計算機環境を有効に利用するための重要な課題である．

我々は，こうした要求を満足すべく Sushi プロジェクトを開始し，その中核である並列分散プログラミング言語 Sushi (Smartly User Schedulable HIgh-level Language) の開発を進めてきた [8, 9]. Sushi の特徴は，問題特有のアルゴリズムを記述する部分と資源管理を行う部分を分離し，後者をプログラム可能なアノテーションとしてユーザに提供していることである．効率はともかくてっ取り早く分散計算環境を利用したいユーザにはデフォルトのアノテーションを提供することにより問題に応じたアルゴリズムのみを書いてもらい，性能を重視したい熟達したプログラマにはアノテーションプログラミングによって満足の行く性能チューニングを行えるよう設計している．

本稿では，アノテーションプログラミングによる性能チューニングや動的資源管理，およびその実装について報告を行う．ユーザは，アノテーションを利用することによりプロセスの初期配置，動的なマイグレーションなどを指示することができる．刻々と変化する並列分散計算環境においてこうした動的な資源管理は極めて重要である．CPU 負荷の軽減だけでなく通信負荷の削減による全体としての性能の向上をはかることができる．

現在，Sushi の処理系は Solaris 2.4 および SunOS 4.1.3 のワークステーションクラスタ上で実装されている．Sushi プロセスを thread として実装し 1 CPU 内での疑似並列処理の負荷を軽減している．また，ヘテロジニアスな計算機クラスタ上での動的なプロセスマイグレーションを含めた実行を実現するため，仮想コード(中間コード)インタプリタ方式と仮想共有メモリを採用している．また，既存の C ライブラリを呼び出すためのインタフェースも備えている．

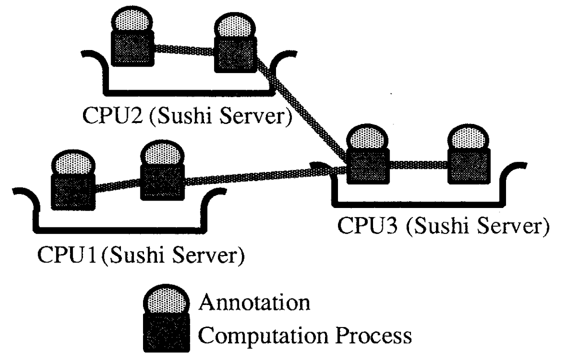

図 1: Sushi プロセス

次節では，計算プロセスとアノテーションによる Sushi のプログラム構造とその利点について述べる．第 3 節では，アノテーションプログラミングの詳細について，具体的な例を題材にして説明する．第 4 節では現在の実装について詳しく報告し，第 5 節で議論と考察を行う．最後に 6 節でまとめを行う．

2 計算プロセスとアノテーション

Sushi プログラムは二つの部分から構成される．**計算部**とアノテーションである．計算部は，問題を解決するための処理アルゴリズムを計算プロセスとストリームによって記述する部分であり，アノテーションは性能チューニングや動的資源管理について計算環境依存の情報を記述する部分である．これら二つを分離する利点は以下のようになる．

- ソフトウェア開発者は，解くべき問題に固有のアルゴリズムの記述を計算部のプログラミングに専念することができ，一方で計算環境に依存した性能チューニングをアノテーションの記述に専念することができる．これは並列分散ソフトウェアの複雑さを軽減し，並列分散ソフトウェア構築の指針を与えることになる．

- 並列分散ソフトウェアの扱いやすさを増す．効率はともかくてっ取り早く分散計算環境を利用したいエンドユーザは計算部のみを記述すれば良い．一方現在の計算環境における性能を重視したいプログラマは，計算部のアルゴリズムを変更することなくアノテーションプログラミングによって満足のいく性能を引き出すことができる．用意されている出来合いのアノテーションを利用することも可能である．

```
1:  main(stdout~) {
2:    spawn generate(2,1000,result~);
3:    spawn primefilter(~result, stdout~);
4:  }
5:  generate(num,limit,result~) {
6:    for (i=num;i<=limit;++i) i >> result~;
7:  }
8:  sieve (~inp, result~) {
9:    n << ~inp;
10:   n >> result~;
11:   spawn sieve(~tmp, result~)@go_neighbour();
12:   while (!eos(x << ~inp)) {
13:      if ( x % n != 0 )
14:         x >> tmp~;
15:   }
16: }
```

図 2: Sushi プログラム例：エラトステネスの篩

- 可搬性と再利用性が向上する．問題固有のアルゴリズムを記述した計算部は，計算環境とは独立であるため容易に他の環境へ移植することが可能である．また，アノテーションも他のプログラムに対して容易に再利用することができる．複数のアノテーションを試験的に付与することによって与えられた計算環境での性能チューニングを効果的に行うことが可能になる．

アノテーションは，一つの Sushi プロセスに一つ割り当てられ，プロセスの実行制御を行う．図 1は，Sushi プロセスとアノテーションのペアが複数 CPU 上で分散している様子を図示したものである．

2.1 計算プロセス

Sushi の計算プロセスは，ストリームで連結された逐次プロセスであり，各 Sushi プロセスは並列に実行される．Sushi の基本構文は C を基にしており，C の制御構造はすべて持っている．ただし，Sushi のデータ構造は文字列と配列のみであり，C が持つようなポインタ，構造体などを持たない．perl に似たタイプレスの言語である．

プロセス間通信はすべてストリームを用いて行われる．UNIX の標準入出力 (stdin , stdout , stderr) もストリームとして与えられている．また，ファイルへの入出力もストリームを通して行われる．Sushi のストリームは複数プロセスによる書き込みと複数プロセスからの読み込みを許しており，同一プロセスからの書き込み順序は保証されるが異なるプロセスからの書き込みについては順序は不定である．新たな Sushi プロセスを生成するためには spawn 文を使い，@ を用いることでアノテーションを Sushi プロセスに結合することができる．

```
1:  annotation go_neighbour(){
2:    init:{
3:       if (load(MY_CPU) > load(MY_CPU+1)) {
4:          spawn_at(MY_CPU+1);
5:       } else {
6:          spawn_at(MY_CPU);
7:       }
8:    }
9:    if (load(MY_CPU) > HEAVY) {
10:      neighbour = neighbours(MY_CPU);
11:      for (i in neighbour) {
12:         if (load(i) < load(MY_CPU)) {
13:            migrate(i);
14:         }
15:      }
16:   }
17: }
```

図 3: アノテーションの例：go_neighbour

図 2は素数生成 (エラトステネスの篩) を行う Sushi プログラムの例である．図の例では，`stdout~`, `~inp`, `result~` 等がストリーム変数である．~ を左側に持つ変数が入力ストリーム，右側に持つ変数が出力ストリームである．入力ストリームからの変数への入力は `<<` オペレータによって表され (例： `n << ~inp`)，出力ストリームへの出力は `>>` オペレータによって表される (例： `n >> result~`)．メインルーチンは `generate` と `primefilter` の 2 つの Sushi プロセスを spawn している．11 行目では，プロセス `sieve` をアノテーション `go_neighbour` 付きで spawn している．

2.2 アノテーション

アノテーションは 2 つの部分から構成される．一つは init 部，そして動的評価部である．init 部はアノテーションが付加されたプロセスの生成時に一度だけ実行され，動的評価部はプロセスの実行中繰り返し実行される．図 3は，先の例にでてきた `go_neighbour` のプログラム例である．`init:` とラベル付けされたブロックが init 部，次のブロックが動的評価部である．例では，init 部で，プロセスが起動されている CPU (`MY_CPU`) と隣の CPU (`MY_CPU+1` , CPU は Sushi 処理系内において一定の順序付けをされている) の負荷を比較し，低い方の CPU を初期 CPU として選択しそこで起動される．動的評価部では，定期的に現

CPU の負荷を評価し一定の値 (HEAVY) よりも大きければ，より小さな負荷の CPU を探してそこへ動的にプロセスマイグレーションを行う．

特に指定しない場合，アノテーションの動的評価部は計算プロセスの実行とは独立に実行される．すなわち，図 3 における負荷の評価のタイミングはプロセス sieve の実行のタイミングには無関係である．実装レベルでのアノテーションの実行については 4 節で詳述する．次節では Sushi の実行とアノテーションプログラミングについてさらに説明する．

3 アノテーションプログラミング

この節では，ワークステーションクラスタ上の現在の実装に基づいて，アノテーションプログラミングと Sushi プログラムの実行について述べる．

3.1 アノテーションのシステム関数

アノテーション部は，計算部からはアクセスを許さない処理系やＯＳが管理する情報を，限定した形でプログラムからアクセスできるよう提供している．表 1 に，アノテーションで利用可能なシステム変数とシステム関数がまとめられている．これらを使うことにより，CPU の負荷，通信負荷，現在の計算機／ネットワーク環境などについて動的にプログラムから認識することができ，プロセスマイグレーションやプロセスの優先度を指定することができる．

システム関数の内，migrate/spawn_at/set_priority はプロセスの実行状態を変更するが，他の関数は現在の環境の動的な認識を行うものである．プログラムから許される変更を最小限に抑え，効率良く実装した組み込みの関数を多数用意することによって，アノテーション実行の負荷を抑えながら柔軟なプログラミングを許すことができる．

3.2 プロセスからアノテーションへのデータ渡し

計算プロセスからアノテーションへのデータ渡しは，現在の実装では値呼びによる引数を渡すことができるのみである．値として CPU 番号や，ホスト名を渡すことで計算プロセスレベルから生成される CPU を指定することが可能である．図 4 では，アノテーション machine_name にマシン名を引き渡すことによってそのマシンで検索プロセスを生成することを指示している．KL1 のプラグマのアノテーションによる実現である．さらに，複数のマシンを指定することにより，プロセス実行中に必要になれば別のマシンへマイグレートさせることも可能である．プロセスが移動しながら必要な仕事をするリモートプログラミングの Sushi による実現となっている．

```
do_query (~inp, outp~) {
  databases["host1"] = "db1";
  databases["host2"] = "db2";
            . . .

  databases["hostn"] = "dbn";

  query << ~inp;
  for (host in databases) {
     spawn get_data(query, databases[host], tmp~)
                        @ machine_name(host);
  }
  spawn regulate(~tmp, outp~);
}

annotation machine_name(name){
    init:{
        spawn_at(hostid_byname(name));
    }
}
```

図 4: 分散データベース検索

```
/* From a process description */
  spawn procees(stream~) @ ann(~stream);

annotation ann(~inp) {
  if (data = top(~inp)) {
     if (load(MY_CPU) > HEAVY) {
          . . .
}}}
```

図 5: アノテーションへのストリーム通信

まだ未実装ではあるが，計算プロセスからアノテーションへの通信を許すことにより，アノテーションと計算プロセスの実行の同期を細かく指定することが可能になる．計算プロセスとは独立に評価実行されていたアノテーションの動的評価部をプロセスの実行に同期させることができる．図 5 の例では，プロセスとアノテーションはストリームで通信をし，アノテーションの動的評価部では入力ストリームにメッセージ (データ) が届いている時のみ負荷の評価を行っている．

3.3 通信負荷の削減

マイグレーションの機能を利用することにより，プロセス間の通信負荷を削減することができる．リモートプロセスとの通信負荷が重い場合，プロセス自体

システム変数	MY_CPU MY_PID	現在の CPU プロセス ID
システム関数	spawn_at(CPU) migrate(CPU) set_priority(PRI) neighbours() load(CPU) traffic(CPU) used_cpu() get_cpu(PID) get_cpu_speed(CPU) get_network(CPU) get_net_speed(NET)	CPU でプロセスを生成 CPU へのマイグレーション プロセスの priority を PRI に変更 ネットワーク上の隣接する cpu 配列 CPU の負荷 CPU と現 cpu 間の通信負荷 現在利用されている cpu 配列 プロセス PID がいる CPU CPU のスピード (SPEC-INT 値) CPU が属するネットワーク ネットワーク NET の伝送速度

表 1: システム変数，システム関数

```
annotation cooperation () {
   if (load(MY_CPU) > HEAVY) {
     neighbouring_hosts = neighbours(MY_CPU);
     for (i in neighbouring_hosts) {
        if (traffic(i) > HEAVY_TRAFFIC) {
           migrate(i);
        }
     }
   }
}
```

図 6: 通信負荷削減アノテーション

```
/* {"hostname", SPEC-INT} */
CPU[]={{"olive",88},{"purple",88},{"ivory",88},
       {"cyan",40},{"magenta",60}};

/* {"networkname", bandwidth (Mbps)} */
NET[]={{"fddi0",100},{"ether1",10}};

/* {"absolute-lib-path",...} */
ANN[]={"/usr/local/lib/sushi/ann/cooperate.so",
       "/usr/local/lib/sushi/ann/another.so"};
```

図 7: コンフィグレーションの記述

がリモートに移動することによって全体としての負荷を軽減できる可能性がある．図 6に，隣接する CPU とのトラフィックを評価し必要ならプロセスに動的に移動を行わせるアノテーションの例を示す．このアノテーションを用いることにより，ネットワーク上を動的に移動させながら通信負荷を抑えるような通信用エージェントを実現することができる．また，パラメータ (HEAVY_TRAFFIC 等) を変更することによって柔軟な制御が可能になる．

この例が示しているのは単純な負荷分散というよりは，コントロールされた負荷の集中である．高性能のワークステーションをネットワークで結合した分散環境においては，全体のパフォーマンスは個々の CPU の負荷よりは通信ボトルネックに依存するといえる．この場合，必要に応じてプロセスが移動することによって通信負荷を軽減することは有効に働く可能性が高い．

3.4 アノテーションライブラリとコンフィグレーション

アノテーションは，Sushi プログラムの記述とは独立に共有ライブラリ化しておくことが可能である．アノテーションライブラリを複数用意し，実行時に動的にバインドさせることにより性能チューニングを効果的に行うことができる．実行時のバインドは次に述べるコンフィグレーションの記述を使って行う．

コンフィグレーションは，実行時に Sushi 処理系に対してグローバルな情報を与えるために利用される．Sushi の処理系はプログラム起動時にコンフィギュレーションファイル `sushi.conf` を読み込み，使用する CPU やアノテーションライブラリを特定する．CPU 情報など，アノテーション内に直接記述することができる情報もあるが，コンフィグレーションによる間接的な記述を行うことによって実行時の性能チューニングを細かく柔軟に行うことができる．

図 7 に `sushi.conf` の例を示す．`CPU[]` に CPU のホスト名と SPEC-INIT の値のペアを記述し，`ANN[]`

に使用するアノテーションライブラリを指定している．また，NET[] には使用するネットワークの名前とバンド幅を指定しておく．各 CPU は，CPU[] に指定された順序でアノテーション内において ID 付けされる．実行時にCPU[] を変えるだけで，使用する CPU の変更を行うことができる．

4 実装

Sushi プログラムを起動するとマスターと呼ばれる Sushi サーバが生成され，Sushi プロセス／ストリームの管理，スケジューリングなどローカル情報の管理の他，次に述べる仮想共有メモリの管理，デッドロックの検出などを行う．リモートマシン上のサーバはスレーヴと呼ばれ，プロセス生成やマイグレーションの要求でオンデマンドで起動される．各スレーヴサーバは，プロセスやストリームなどローカルな情報の管理においてはマスターサーバと対等に振舞う．以下この節では，Sushi サーバの実装について詳述する．

Sushi の実装方針は，ネットワーク化されたヘテロジニアスな計算機クラスタ上で，実行中の任意の Sushi プロセスを適度にマイグレートしながら全体の計算を進めることを可能にすること，そしてネットワーク上の Sushi プロセス間を高速なストリーム通信で結び付けることである．

4.1 仮想コード（中間コード）方式

ネットワーク上のヘテロジニアスな計算機環境に適する専用の仮想コード（中間コード）を設計した．このコードは，通常のスタック上の計算をベースに，プロセスの生成，消去，ネットワークを介した通信を含むストリーム通信のためのプリミティブで拡張している．将来的に，この仮想コードをネイティブコードに変換するコード生成器を作る計画で，プログラムの部分部分を単位とするネイティブコードによる高速化が可能となる．

4.2 Sushi プロセス

1 個の Sushi プロセスが必要とする計算機資源は非常に小さいので，Sushi の実現ではマルチスレッド処理を採用した．一つの仮想コード・インタープリタで一つの Sushi プロセス（スレッド）を処理することとし，スケジューラという特別なスレッドで，タイマー割り込み，入出力のためのコンテキスト・スイッチ，プライオリティ・キューの処理をすることで，複数のインタープリタを疑似的に並列処理している．この方法では，Sushi プロセス間の実行順序は完全に自由にでき，実行順序に実装上の制限がない．（マルチスレッド処理を採用したことにより，プログラムから呼び出すことができるユーザ定義の C ライブラリもマルチスレッドで動き，Sushi を簡易なマルチスレッド化パッケージとして使うこともできる．）

4.3 プロセスマイグレーション

Sushi は，ネットワーク上で複数の計算機を使う計算をターゲットとしているので，リモートの計算機で Sushi プロセスを起動する，あるいは，現在動いている Sushi プロセスをリモートの計算機にマイグレートする機能が必要である．この実現には，次節の仮想共有メモリの仕組みを採用した．この方式では，コードとヒープ領域のデータ（セル）はマイグレーションの度に毎回転送する必要がなく，Sushi プロセスのスタック領域と仮想コードインタプリタが使っている数個のレジスタの中身だけマイグレーションの度に転送するだけである．スタック領域が仮想共有メモリの対象でないのは，リード／ライトが一つの CPU 上でだけ起こるので，仮想共有メモリのような大掛かりな仕組みでは効率が悪くなってしまうためである．

4.4 仮想共有メモリ

プロセスの実行途中での動的なマイグレーションを可能とするためには，仮想コードをネットワーク上でポジション・インデペンダントにしておかなければならない．そのためには，仮想コードに表れる全てのメモリアドレスを仮想的に全ての計算機で共有させる仕組みが必要である．しかし，Sushi では仮想共有メモリ [2] の一部の機能しか必要としないので，これを簡略化して効率の良い方法で実現している．コード領域は，各計算機上で最初の Sushi プロセスが動き出す直前にリザーブされたメモリ空間にオンデマンドでコピーしている．ライトを伴うメモリ空間であるヒープ領域は，セルという可変長のライト・ワンスのメモリ単位で管理することとし，全てのアドレスへのアクセスではなくセル単位でのアクセスに関してのみ仮想共有メモリを構築した．通常の仮想共有メモリでは，全ての CPU でセルデータのキャッシュの一貫性を保つ仕組みが必要であるが，セルはライト・ワンスなので Sushi の実現ではキャッシュの一貫性を保つ仕組みを省略できる．その他，インタープリタが内部的に利用するメモリ空間，Sushi プロセスのスタック空間，ストリームのバッファなどは全てローカルな計算機上に取ることができるので，仮想共有メモリの対象から外している．このため，Sushi の仮想共有メモリの対象は狭く，そのオーバヘッドは小さくなっている．

4.5 アノテーション

スケジューラは，単にSushiプロセスの実行順序を制御するだけでなく，アノテーションの実行も担当する．各Sushiプロセスのアノテーションの評価は，最初の一回と後は適当な時間間隔で何度も実行を繰り返すだけでその都度実行が完結し，前回の実行終了状態を保存しておく必要がない．このためアノテーションに対してSushiプロセスのようにスレッドを割り当てる必要がなく，単に一時的にスタックの一部を利用するだけで良いので，アノテーションは一番オーバヘッドの少ないスケジューラの内部で計算させている．実際の実現は，スケジューラからSushiプロセスへコンテキストを入れ換える直前に，スケジューラがそのプロセスに結び付けられているアノテーションを関数呼び出しの形で実行している．

4.6 ネットワーク透明なストリーム入出力

ネットワークを介したSushiプロセス（スレッド）間のストリーム通信を高速に行なうためには，シグナル(SIGPOLL,SIGIO)を使った割り込み処理が必要である．実行中のスレッドから強制的に制御を奪い取り，通信ルーティンを呼ぶためである．ただし，割り込みのトラップは，時に検出に失敗する場合があるので，頻度の低い定期的なポリングを併用している．しかし，それでもネットワーク上で通信の同期を取るコストは大きいので，通信のスループットを上げるためにストリームに流すデータをスレッド単位でバッファリングしている．Sushiの実装では，全てのバッファはネットワーク上でポジション・インデペンダントでないローカルメモリを使っているので，プロセスの実行中にマイグレーションが起こった場合，このバッファをプロセスの移動した先の計算機に転送しなければならない．しかし，プロセスが移動しても必ずしもバッファリングしているストリームを使った通信をするとは限らないので，移動した後の最初のストリームへの入出力をトラップして，バッファの転送を行っている．

5 議論と考察

5.1 言語デザイン

アノテーションは，通常はユーザが知る必要のない処理系やOSが管理する情報に，特定の仕方でプログラムからアクセスを許す機構である．これは，リフレクション[5, 10]におけるメタレベルの記述を限定した形で提供するものと言うことができる．プロセスの実行制御を行うという意味では，並列論理型言語 KL1 [7]のプラグマを一般化し，プログラム可能な形でユーザに与えているものと考えることもできる．また，Sushi の動的資源管理という点では，PCN[3]のような静的な資源管理情報をユーザプログラムとは独立に記述できる言語に対する拡張になっている．

当初，Sushi の言語デザインとしては，計算部とアノテーションの完全な分離を前提としていた．つまり，アノテーションの動的評価部の実行は計算プロセスの実行とは完全に独立であった．これはユーザが意識せずともアノテーションによって最適な資源配分を行いたいという目的のためであった．

しかし一方で，アノテーションの実行を計算プロセス側から制御したいという要求も生まれてきた．また，複数プロセスの協調をアノテーションを用いて記述する場合にもプロセスとアノテーション間の通信は有効である．ただしこれは，計算プロセスとアノテーションを分離する当初の選択に反するものであり，アドホックな拡張は避けなければならない．現在，我々はアノテーションの役割について再考察しており，計算プロセスとアノテーションの最良な関係をデザインするべく検討を続けている．

5.2 実装

Sushiの実装での仮想コードの採用は，ネイティブコード方式に比べて実行性能の低下はあるが，通常のインタープリタ方式の実装と同様にコンパイルとライブラリのリンクに掛かる時間が少ないというメリットがある．また，実行コードを予め各計算機に配っておく必要がなく，計算の実行中に必要に応じてリモートの計算機にコードが自動転送できるメリットがある．一般に，ネットワーク上のヘテロジニアスな計算機環境ではネイティブコードを使った計算は繁雑で，ユーザが理解して管理しなければならないことが多くなってしまう．そのために，エンドユーザのための言語であるSushiの実現には適さないと判断した．

Sushiプロセスの実現にマルチスレッド処理を採用しているのは，主に時分割方式のフェアなプロセスの実行制御をするためである．また，共有メモリ型のマルチCPU計算機での高速実行も考慮している．他の選択として，一つのインタプリタで複数のSushiプロセスを実行する方式も考えられるが，この方法では組み込み関数，ユーザ定義のC関数の呼び出し中のSushiプロセスの実行のタイマによる中断ができない．また，共有メモリ型のマルチCPU計算機で複数のCPUを同時に使った高速計算ができなくなる．

Sushiのプロセスマイグレーションは，プロセス実

行のどの時点においてもどの計算機に対しても可能であるので，プロセスから直接アクセス可能なメモリ空間は，全ての計算機で共有していなければならない．プロセスの生成が別の計算機上で可能であるだけで良い言語の場合は，メモリ空間の共有が必須でないため，このためのオーバヘッドを避けることができる．OS レベルでの仮想共有メモリを使った場合，このオーバヘッドを削減する手段は限られてしまうが，Sushi 言語のレベルでの仮想共有メモリは，大幅な機能の限定とその限られたタイミングでのアクセスだけに対処すれば良いので，そのオーバヘッドを大幅に削減することが可能である．実際，Sushi の実行中にリモートのメモリにアクセスするのは，ほとんどの場合，バッファリングされたデータをリモートのストリームへ書き込むときと読み出すときだけである．

5.3 形式モデル

計算プロセスは逐次プロセスの並列実行に他ならず，CSP [1] スタイルの意味論を与えることが可能である．また，多入力多出力ストリームの動作を並列制約言語で記述したものに [6] がある．

しかし，分散計算機環境上でのマイグレーションなどアノテーションの機構を含めた Sushi プログラムの全体的な挙動を形式的に記述することは難しい課題である．分散 CCS を用いて，Sushi の挙動をモデル化する試みも行われているが [4]，一部の機能のみを対象としており完全なモデル化ではない．しかし並列分散プログラミングの可能性が広がるにつれ，プログラミング環境としてのシミュレータなど必要性は増しており，形式的なモデル化の重要性は高まっている．

6 おわりに

本稿では，並列分散プログラミング言語 Sushi のアノテーションによる性能チューニングや動的資源管理について報告を行った．プログラムを計算プロセスとアノテーションに分離することにより，分散計算機環境での柔軟な資源管理を扱いやすくユーザに提供している．これはまた，並列分散ソフトウェアの可搬性と再利用性を向上させている．

現在，Sushi は第一版が Solaris 2.4 および SunOS 4.1.3 のワークステーションクラスタ上に実装されており，言語デザインや実装の面から評価改良を行っている．並列分散ソフトウェア開発を効果的に行うための言語／開発環境として，どのような技術が本質的であるのかが問われる課題であり，Sushi のアノテーションやプロセスマイグレーションはその中の重要な要素技術であると考えている．

謝辞

本研究を進めるにあたり貴重な意見をいただいた神田陽治氏，蓬莱尚幸氏，和田裕二氏に感謝する．

参考文献

[1] C.A.R.Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[2] D. Lenoski et al. The directory-based cache coherence protocol for the dash multiprocessor. In *IEEE proc. 17th Int'l Symp. Computer Architecture*, 1990.

[3] I. Foster, R. Olson, and S. Tuecke. Productive parallel programming. *Scientific Programming*, Vol. 1, pp. 51–62, 1992.

[4] G. Hains and H. Sugano. A process-algebra model of migration. Submitted to HICCS-29, 1995.

[5] B. C. Smith. Reflection and semantics in lisp. In *Proc. 11th ACM Symposium on Principles of Programming Languages*, pp. 23–35, 1984.

[6] H. Sugano. Stream-based Programming in Concurrent Constraint Computation. In *11th Conference Proc. of JSSST*, 1994.

[7] K. Ueda and T. Chikayama. Design of the Kernel Langugae for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6, pp. 494–500, 1990.

[8] 鵜飼孝典、山中英樹、上田晴康．プロセスの再配置可能なストリームベース並列プログラミング言語．情報処理学会プログラミング研究会報告 94-PRG-18, Vol. 94, No. 65, pp. 161–168, 1994.

[9] 山中英樹、鵜飼孝典、上田晴康、菅野博靖、和田裕二．Sushi – プロセスの再配置可能な並列分散プログラミング言語 –．情報処理学会第回全国大会, 1994.

[10] 渡部卓雄．リフレクション．コンピュータソフトウェア, Vol. 11, No. 3, pp. 5–14, 1994.