九州大学学術情報リポジトリ
Kyushu University Institutional Repository

FUCE上のストリーム処理とその記述言語

長谷川，隆三
九州大学大学院システム情報科学研究院知能システム学部門

藤田，博
九州大学大学院システム情報科学研究院知能システム学部門

雨宮，聡史
九州大学大学院システム情報科学府知能システム学専攻 ： 博士後期課程

越村，三幸
九州大学大学院システム情報科学研究院知能システム学部門

他

https://doi.org/10.15017/1516220

出版情報：九州大学大学院システム情報科学紀要. 11 (1), pp.31-38, 2006-03-24. 九州大学大学院システム情報科学研究院
バージョン：
権利関係：

九州大学大学院
システム情報科学紀要
第11巻 第1号 平成18年3月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.11, No.1, March 2006

FUCE上のストリーム処理とその記述言語

長谷川隆三* ・ 藤田 博* ・ 雨宮聡史** ・ 越村三幸* ・ 雨宮真人*

Stream Processing on FUCE and Its Implementation Language

Ryuzo HASEGAWA, Hiroshi FUJITA, Satoshi AMAMIYA,
Miyuki KOSHIMURA and Makoto AMAMIYA

(Received December 9, 2005)

Abstract: In this paper, methods of concurrent programming on FUCE machine and language designs to implement them are studied. The idea of multi-thread processing on FUCE is far different from the others fitted to conventional computer systems. We have been developing a family of languages that are suitable for writing programs which should be executed efficiently on FUCE. The language features include several ways for handling synchronization, mutual exclusion, and stream processing. These should enable us to write more FUCE friendly programs, and a unique FUCE-OS in particular.

Keywords: Stream processing, Fine-grain thread processing, Petri net, Concurrent programming, FUCE architecture

1. まえがき

FUCEは，応用プログラムからOSに至るまで，あらゆる処理を“走り切り”スレッドを基本単位として実行することを特徴とする斬新なアーキテクチュアである．近い将来，現行のフォンノイマン型コンピュータに訪れるであろう性能向上の行き詰まりを打破する方式の一つとして期待されている．これまでに，いくつかの逐次的なプログラムがFUCE上でマルチスレッド化され，効率よく実行されることが確認されている．しかしながら，OSのような本質的に並行的なプログラムに関しては，プログラミング手法ならびに言語処理系等が未整備なため，FUCEの有効性が十分には実証されていない．

一般に，並行プログラミングシステムにおいては共有資源に対する排他的制御が重要な問題であり，モニタのような高度な機構がソフトウェア的に用意されることが多い．しかし，最終的には test&set のようなアトミック命令を使うハードウェア的な解決に帰着される．FUCEは，スレッドという同時実行可能なコード群を扱うのだから，すでに並行プログラミングに必須な機構が備わっていると考えられる．実際，fan-in数に基づいて実行可能なスレッドを判定する機構として，並行スレッド間の同期機構がハードウェア的にサポートされている．具体的には，fan-inカウンタの排他アクセスのためにlock/unlock命令が用意されている．

FUCEのスレッドを単に関数のコード断片として見ると，fan-in数とはその関数が必要とする入力データの個数にすぎない．したがって，スレッドが実行対象として選ばれ，fan-inカウントが初期値に設定された後は，要求データ到着とともに単調減少する一方であり，これが0になった時に実行可能と判定されて待ち行列に投入されるわけである．fan-inカウント値が非単調に増減したり，負値や初期値を超える値になることはない．我々の並行プログラミングにおいては，このfan-inカウンタ自体をあたかもセマフォのように利用することを考える．

我々は，一般のCプログラムをFUCE機械語に変換する言語処理系を開発中であるが，並行処理，特にストリーム処理向けの機能については，独自の記述スタイルを検討してきた．それは，FUCEアーキテクチュアの“走り切り”スレッド処理の機構が，“継続に基づく中断(suspension)なしの並行タスク”という新たな並行処理の枠組みを可能とするからである．この新しい枠組みに基づいた記述スタイルは，FUCE中間語IMLの拡張構文で実現することができた．本稿では，拡張IMLとこれに基づくストリームプログラミング手法について述べる．

2. FUCE中間言語IML

IMLは本来，C等のユーザ言語からFUCE機械語への変換過程における作業用言語という位置づけである．しかしながら，FUCE依存コードを適度な抽象度で扱いたいFUCE-OS等の開発者にとっては，そのままユーザ言語となるべきものでもある．そのため，IMLは基本的にANSI標準のC言語をほぼそのまま踏襲し，これに若干の変更と拡張を施すことによって実現されている．拡張構文の概要を**Table 1**に示す．

平成17年12月9日受付
* 知能システム学部門
** 知能システム学専攻博士後期課程

Table 1 IML syntax as an extension of C syntax.

```
<function/process definition> ::=
        <type specifier> ( function | process )
        <function/process name>
        ( <formal parameters> ) [ < <fan-in> > ]
        { <function/process body> }
<thread definition> ::=
        thread <thread name> [ < <fan-in> > ]
        { <thread body> }
<DA variable definition> ::=
        darea <variable names> ;
<DA variable assignment> ::=
        <DA variable> := <expression> ;
<thread transfer> ::=
        => <thread name> ;
<function call> ::=
        <DA variable> := <function name>
        ( <parameters> ) => <thread name> ;
<self-recursive call> ::=
        recur ( <parameters> ) ;
<statement to be executed last> ::=
        return; | <thread transfer> |
        <self-recursive call>
```

2.1 チャネルに関する拡張

ストリーム処理においては，データの送り手と受け手との間に，データを仮置きする場所を設けることが多い．それは，通常は有限長の待ち行列形式のバッファである．ここでは，**チャネル**と呼ばれる基本的にデータ1個分の容量の中継記憶領域を用いることにする．

チャネル構造体

```
struct chan {int flag, value, from, to;}
```

flagが0 のとき，ストリームが閉じられていることを表す．value は1個のストリーム要素（ここでは整数値）を格納する．from は送出元の，to は受取先のプロセスの入口スレッド番号を格納する．

プロセス定義

```
process P(chan *ch, ...) <2> { ... }
```

*chは，チャネル構造体へのポインタ．<2> は，Pの入口スレッドのfan-in数．無指定の場合，<1> とみなす．{ ... } は，Pの本体である．

インスタンス生成

```
int p = new P();
```

Pのインスタンスを生成し，データ領域とを確保する．

ストリームの敷線

チャネル構造体の from メンバにストリームデータ送出元のプロセスIDを，to メンバに受取先のプロセスIDをそれぞれセットする．

プロセスの起動

プロセスの起動時に，チャネル名を実引数として呼出す．このとき，ストリーム送出側なら<+> 受取側なら<->のモード指示子を付す．

Table 2 Extended syntax for channel.

Channel syntax	Meaning
ch !! v; ch.xx !! v;	write with notification
ch !!! v; ch.xx !!! v;	write without notification
ch !! nil;	close with notification
ch ?? y; ch.xx ?? y;	read with notification
ch ??? y; ch.xx ??? y;	read without notification
ch ?= nil	true if ch is closed
ch |> q;	change receiver
p(cha::ch1<*>, chb::ch2<+>);	(re-)connenct channel

例： p(ch1<+>, ch2<->); q(ch1<->, ch2<+>);

このほかのチャネル操作に関する拡張構文については，**Table 2**に示す．

3. ストリーム処理の実現方式

スレッドの継続命令(cont)に伴うfan-inカウントの機構とチャネル構造体を基本的な道具とし，それらの組み合わせ方によって異なる4種のストリーム処理方式を示す．

3.1 継続方式

書き手プロセスwが，ストリーム要素xの値を生成し，読み手プロセスrに伝える動作を永続的に繰り返すストリーム処理のIML記述を**Fig. 1**に示す．ストリーム要素はチャネルchを中継してwからrへ伝えられる．プロセスw,rの動作は継続命令contによって制御される．contは意味に応じてinit-/trigger-/ack-contと呼び分ける．

wはinit-cont，あるいはrからのack-contにより起動されると，chにデータxを書込み，rにその読出しを促すtrigger-contを発行する．rはwからのtrigger-contにより起動されると，chからデータxを読出し，wに次の書込みを促すack-contを発行する．w(r)のfan-in数が2であるのは，trigger-(ack-)cont以外に再帰継続のためのrecur-contを必要とするからである．

wとrが同時に動作することはなく，ゆえにchに同時にアクセスすることもない．また，wとrは常に交互に動作し，ゆえにw(r)が連続してchに書き込む（読み出す）ことはない．

この動作は，**Fig. 2**に示すようなペトリネットに準じたグラフで表すと理解しやすい．1個のトークン（黒丸）は，1個のcont命令発行に対応する．プロセスwへ向かう右側のトークンはinit-contに，w,r各々の左のループ上のトークンはrecur-contにそれぞれ対応する．

トランジション（横線）において，その右下に記され

```
process main() {
  w = new W(); r = new R();
  ch->from = w; ch->to = r;
  w(ch<+>); // wのrecur-cont発行
  r(ch<->); // rのrecur-cont発行
  cont w;   // wのinit-cont発行
}

process W(chan *ch) <2> {
  /* xの値を決める計算 */
  ch->value = x;
  cont ch->to; // rのtrigger-cont発行
  recur;
}

process R(chan *ch) <2> {
  x = ch->value;
  cont ch->from; // wのack-cont発行
  /* xの値を使った計算 */
  recur;
}
```

Fig. 1 Continuation method.

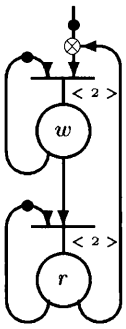

Fig. 2 Petri net for two processes.

たfan-in数に等しいトークンが揃うと，1個の合体トークンとなってプレース（円）に移動し，そこに記された名前のプロセスが起動する．プレースの下方から伸びる出口アークごとに，1個のトークンが発行されることになる．すべてのアークにトークンを発行し終わると，プレースにあったトークンは消滅する．w(r)の左側の出口アークからはrecur-contトークンが，右側からはtrgger-(ack-)contトークンが発行される．w,rのプレースに同時にトークンが留まることがないことが容易に判る．

3.2 たすきがけlock/unlock方式

FUCEには，スレッドのfan-inカウントに関連して，lock/unlock命令というスレッドアクセスに関する排他制御機能が用意されている．これを積極的に利用したプログラミングスタイルを次に示す．

w,rのfan-in数は共に1で，リンク時のrecur-contにより直ちに起動される．w,rは各自のスレッドに対しlockを掛け，処理完了後，相手スレッドをunlockする．w,rは共に，lockに失敗すればrecurによりbusy waitする．wはrがchから読み出さない限り，連続して ch に書き込むことはできず，rもwがchに書かない限り，連続してchから読み出すことはできない．しかし，wの書き込み後にrが読み出すという保障はなく，w,rが同時にチャネルにアクセスする可能性があり，読み出したデータが未定義ということがある．

```
process main() {
  w = new W(); r = new R();
  ch->from = w; ch->to = r;
  w(ch);
  lock(ch->to);
  r(ch);
}

process W(chan *ch) <1> {
  if (!lock(ch->from)) recur;
  /* xの値を決める計算 */
  ch->value = x;
  unlock(ch->to); // rに対しunlock
  recur;
}

process R(chan *ch) <1> {
  if (!lock(ch->to)) recur;
  x = ch->value;
  unlock(ch->from); // wに対しunlock
  /* xの値を使った計算 */
  recur;
}
```

Fig. 3 Lock/Unlock method.

この問題を解決するため，w,rを起動する前に，rに対してlock(r)を掛けることにする．こうすれば，wが初回の書き込みを行ってunlock(r)をしない限り，rは読み出しができないので，wの書き込みとrの読み出しが交互に行われるようになる．

上記プログラム例では，`r(ch)`の直前にrに対するlock（`lock(ch1->to)`）を置いた．もし，その前に置いた`w(ch)`内からの`unlock(r)`がlock(r)より前に実行されると，rはlockされたままになる．これが懸念されるなら，上述の通り，`w(ch); r(ch)`の前に`lock(r)`を置くようにしなければならない．

ここでは，`ch->from`にchの送出元プロセスwを，`ch->to`にchの送出先プロセスrを設定した．プロセスが1入力1出力の場合はこれで良いが，複数入力／複数出力の場合は所望の動作をしない．正しくは，`ch->from = getid(); ch->to = getid();`のように，チャネルの始点，終点毎に新たなスレッドを確保しなければならない．ここで，getid[†1]は，lock対象のスレッド番号を確保する命令である．

3.3 ハイブリッド方式

次に，継続方式とlock/unlock方式を混合した方式を示す．これをハイブリッド方式と呼ぶ．

片方向のlock/unlockとcontを組み合わせている．wの

†1 現在サポートされていない．

```
process main() {
  w = new W(); r = new R();
  ch->from = w; ch->to = r;
  w(ch);
  r(ch);
}

process W(chan *ch) <1> {
  if (!lock(ch->from)) recur;
    /* xの値を決める計算 */
  ch->value = x;
  cont ch->to;
  recur;
}

process R(chan *ch) <2> {
  x = ch->value;
  unlock(ch->from);
  /* xの値を使った計算 */
  recur;
}
```

Fig. 4　Hybrid method.

fan-in数は1であり，一方，rは2であることに注意．wは自身のスレッドに対しlockを試み，失敗すればrecurによりbusy waitする．wがlock(w)を掛けてchに書き込むと，rがchから読み出すまではunlock(w)されない．よって，その間はwが連続してchに書き込むことはない．

一方，rはwからのtrigger-contによってしか起動されず，w,rが同時にchにアクセスすることはない．rが連続してchからデータを読むことはなく，読み出したデータは必ず定義されている．

3.4 Look-in 方式

cont, lock/unlockのいずれにも依らず，ふつうのメモリに置かれたフラッグだけを頼りに相互排他制御を行う方式を次に示す．フラッグは，チャネル構造体中にupdメンバとして追加する．

w,rが同時にchにアクセスすることがある．wはupdが1（既更新）のときはrecurしてupdが0（既読）になるのを待つ．rがchを読まない限りupdが0とならないので，wが連続してchに書き込むことはない．また，rはupdが0のときはrecurしてupdが1になるのを待つ．wがchに書き込まない限りupdが1とならないので，rが連続してchから読み出すこともない．

この方式の不安は，wとrが共通のch.updに書き込みを行う点である．しかし，wがupdが1を設定できるのはupdが0のときであり，この場合はrはrecur待機中である．また，rがupdが0を設定できるのはupdが1のときであり，この場合はwはrecur待機中である．すなわち，w,rは交互にしかch.updを変更できないので安全である．

現在想定しているchは，書き手wも読み手rも唯一である．wがchにデータをひとつ書くと，rがchからそれを読み出す．これが交互に繰り返される．wが連続してchに書き込んだり，rが連続してchから読み出すことは考えていない．

```
struct chan {int flag,value,from,to;
             int upd;}

process main() {
  w = new W(); r = new R();
  ch->from = w; ch->to = r;
  w(ch);
  r(ch);
}

process W(chan *ch) {
  if (ch->upd==1) recur;
    /* xの値を決める計算 */
  ch->value = x;
  ch->upd = 1; // 読出し可に設定
  recur;
}

process R(chan *ch) {
  if (ch->upd==0) recur;
  x = ch->value;
  ch->upd = 0; // 書込み可に設定
  /* xの値を使った計算 */
  recur;
}
```

Fig. 5　Look-in method.

ストリームを分岐させる場合，`p(a::ch1<+>, b::ch2<+>)`のように出力先に応じて個別のチャネルを使用し，ストリームの合流の場合も同様，`q(a::ch1<->, b::ch2<->)`のように入力元に応じて個別のチャネルを使用する．そもそも，セマフォーやlock/unlockは，複数のwが同時に書き込むことを防ぐためにある．したがって，上記のようなchの使い方をする限り，これらの機構は不要である．そこで，look-in方式のようなlock/unlockを使用しない方式が成り立つ．

KL1[3]におけるように，変数が未定義ならsuspendし，変数がinstantiateされた時にreadyとなるようなbind hookを実現することを考えよう．bind hook要求は複数箇所で発生しうるので，これらの要求を受理する（待ち行列につなぐ）操作は，共有資源に対する同時書き込み問題を解決しなければならず，本質的にtest&setを必要とする．開世界では，資源要求元は固定的でないので，`p1(a::ch1<+>); ... pn(a::ch1<+>); q(a::ch1<->);`のように，共有チャネルch1を使用し，w(pi),r(q)間の排他制御をlock/unlockを用いて行う方が便利である．

4. プログラム例

100までの素数を計算するプログラムの記述例を**Fig. 6**に示す．(1)の`prints(ch)`は，chのデータをすべて出力するものとする．効率は落ちるが，(1)の代わりに，`if (a>100) {ch !! nil; exit;}`としてもよい．

(2)では，チャネル送出先変更指令`<*>`を用いた．`sieve`の入力チャネルchは，`sieve`内で生成した`filter`プロセ

```
process main() {
  darea chan ch1;
  int n = new ints();
  int s = new sieve();
  n(i::2,ch::ch1<+>);
  s(ch::ch1<->);
}

process ints(int i,chan *ch) <2> {
  if (i > 100) { ch !! nil; exit; }
  ch !! i;
  recur(i::i+1);
}

process sieve(chan *ch) <2> {
  darea chan ch2;
  int a;
  if (ch ?= nil) exit;
  ch ??? a;
  if (a>10) {prints(ch); exit;} // (1)
  printf(a);
  int f = new filter();
  int s = new sieve();
  f(e::a,cha::ch<*>, // (2)
    chb::ch2<+>);
  s(ch::ch2<->);
}

process filter(int e,chan *cha,
               chan *chb) <3> {
  int a;
  if (cha ?= nil) {
    chb !! nil;
    exit;
  }
  cha ?? a;
  if (a % e == 0) {
    recur[2]; // (3)
  } else {
    chb !! a;
    recur;
  }
}
```

Fig. 6 IML code for a prime number generator.

スのインスタンスfにそのまま引き継がれるが，その際，chの送出先(ch.to)がsieveからfに変更される．これにより，intsプロセスからfilterプロセスへ直接データが送出されるようになる．以後，上位のfilterプロセスから下位のfilterプロセスに対しても，同様のことが実現される．

(3)では，recur-contを2個発行していることに注意．a%eが0のとき，chbに出力されないので，chb.toからのackがこない．したがって，あたかもchb.toにデータを送出したかのように，仮想的にack-contを発行してやる必要がある．

次に，**Fig. 7**に併合プロセスのペトリネット，**Fig. 8**にそのIML記述を示す．mergeプロセスrは，偶数列を生成するプロセスpからのストリーム(cha)と奇数列を生成するプロセスqからのストリーム(chb)を併合し，一本の出力ストリーム(chc)にして計算プロセスsに引き渡す．

mergeはp,qからのtrigger-contにより起動され，原則的にcha,chbを交互に見る．iはこのための制御変数

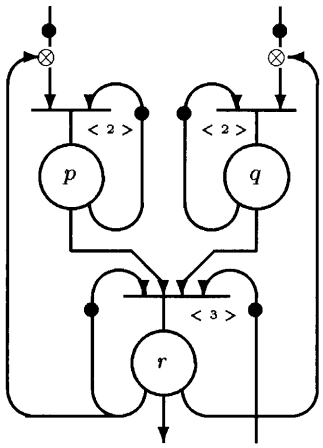

Fig. 7 Petri net for merge process.

である．cha(chb)を覗いて新データが届いていれば(cha.upd(chb.upd)==1)，そのチャネルからデータを読み出す．届いていなければ(cha.upd(chb.upd)==0)，他方のチャネルに届いている筈だから，chb(cha)よりデータを読み出す．

5. ストリーム処理の効率改善

チャネルがデータ1個分のバッファしか用意していないために，書き手も読み手も互いに相手の処理中は動作できない，という状況に陥りやすい．これでは全体のスループットが低く，満足な性能が得られないという場合がある．このような場合に，チャネルを適宜追加することによって改善を図る手法を示す．

5.1 交代バッファ

たとえば，**Fig. 9**に2つのチャネルを交代バッファとして使用する例を示す．図では，一見スレッド実体も2倍必要であるかのようであるが，実は一つのスレッドがモード変化（Aバッファ使用モードとBバッファ使用モード）をしているだけである．書き手がA(B)バッファに書いている間，読み手はB(A)バッファから読むことができる．

Fig. 10にタイミングチャートを示す．(a)は単一チャネル，(b)が二重化チャネルの場合である．A,Bはwからrへの通信に使用されるチャネル名を表す．*tc*は書き手プロセスwから読み手プロセスrへのtrigger-contを，*ac*はrからwへのack-contを，*mc*はwにおけるAからBへのモード変更のためのtrigger-contを，それぞれ表している．rにおいて，Bモード実行終了時にwへのack-contが不要であることに注意．このほかに,w,rプロセスは自身への継続のため，recur-contを発行しているが，図には表示していない．

こうして，書き手と読み手との処理速度に大差がなければ，どちらも間断なく稼動することができる．

```
struct chan {
  int flag, value, from, to, upd;
}

process main() {
  darea chan ch1,ch2,ch3;
  int p = new evenodd();
  int q = new evenodd();
  int r = new merge();
  int s = new comp();
  p(i::0,ch::ch1<+>);
  q(i::1,ch::ch2<+>);
  r(cha::ch1<->,chb::ch2<->,
    chc::ch3<+>);
  s(ch::ch3<->);
  cont p;
  cont q;
}

process evenodd(int i,chan *ch) <2> {
  if (i > 10) { ch !! nil; exit; }
  ch.upd !!! 1;
  ch !! i;
  recur(i::i+2);
}

process merge(chan *cha,chan *chb,
              chan *chc) <3> {
  darea int i=0;
  int u,v;
  if (i==0) cha.upd ??? u;
  else chb.upd ??? u;
  if (i != u) {
    cha.upd !!! 0;
    cha ?? v;
    chc !! v;
  } else {
    chb.upd !!! 0;
    chb ?? v;
    chc !! v;
  }
  recur(i::(i+1)%2);
}

process comp(chan *ch) <2> {
  int v;
  ch ?? v;
  printf(v);
  recur;
}
```

Fig. 8　IML code for a merge process.

5.2 チャネルリスト

書き手と読み手の速度差が大きい場合には，交代バッファを用いても，速い方が遊んでしまう．そのような場合，2個以上のチャネルを動的に生成して速度差を吸収するという方法がある．

たとえば，素数計算において，*sieve*が素数pを受け取る度に，これをただちに*outs*プロセスに引き渡したい．この通信は一方向で，*sieve*は*outs*からの*ack*を待ちたくない．このような場合，*sieve*と*outs*は1個のデータ授受毎に新しいチャネルを使用する．この様子を**Fig. 11**に示す．s_i, r_iはそれぞれi回目の再帰フェーズの*sieve*および*outs*プロセスを表す．ch_iはs_i, r_iが使用するチャネルである．*sieve*はチャネルにデータを書込み*outs*に送出する．

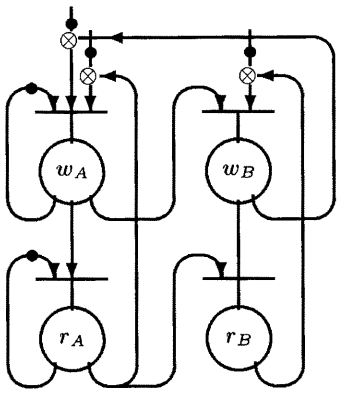

Fig. 9　Two processes communicating via alternating buffers.

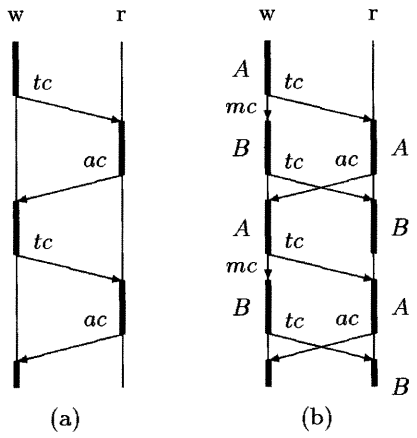

Fig. 10　Timing chart for two communicating processes.

*outs*はデータが送られてくるとチャネルからデータを読出すが，送出元の*sieve*に*ack*を返さない．

ここで，送られてくる*cont*要求の数mがfan-in数nを超えた場合，残る$m-n$個の*cont*要求は捨てられないことが重要である．

チャネル構造体は次のように変更する．

```
struct chan { int flag, value, from, to;
              chan *next;}
```

すなわち，通常のメンバのほかに次のチャネル実体へのポインタである*next*メンバを付加している．要するに，チャネルのリスト構造を構成する．こうして，生産者と消費者の間に可変長バッファを置いて，両者の処理速度差を吸収することができる．

このように，より強力に速度差を軽減できるメカニズムを採用すれば，より徹底したデータ駆動型計算が実現されることになる．ところが，生産者が消費者の要求より一方的に速い場合，過剰生成という問題が生じる．逆に，消費者の速度が生産者側より速い場合，チャネルすら未定義となって，より深刻な問題となる．これには以下に示すように，いくつか対処法が考えられる．

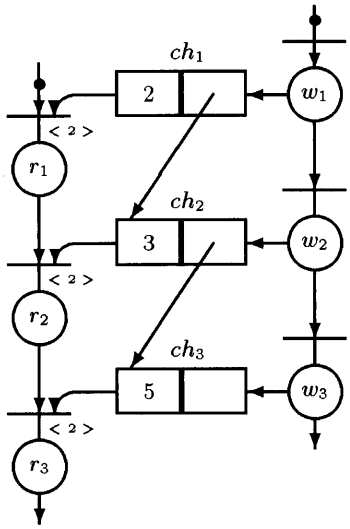

Fig. 11 Channel List.

対処法 1

w_iプロセスとr_iプロセス間の通信チャネルをch_iとする．$ch_i.flag$: $undef$ならば，r_iは$ch_i.to$に自thidを記入し，$ch_i.from$ (w_iのthid)に対し$cont$を発行し，w_iにデータを要求する．w_iは$ch_i.to$ (r_iのthid)が記入されていれば，データを書込んだ後，そのthidに対し$cont$を発行し，r_iを起動する．未記入ならば，データを書き込むだけである．

この場合，2つのプロセスが同じチャネルフィールドに読み書きを行おうとするので，セマフォーやtest&set命令が必要になる．

対処法2

contカウントのみで同期制御する．w_iプロセスはch_iにデータを書き込む度に，r_iにtrigger-contを発行する．r_iはtrigger-contを受けて，ch_iからデータを読み出す．この方法はtest&set命令を必要としないが，書き手が読み手のthidをどのように入手するかが問題となる．

1. 通常の再帰使用

   wプロセスもrプロセスも再帰の度に新たにthidがとられ，新チャネルch_iが生成される．しかし，このch_iを受ける相手r_iのthidは不明なので，trigger-contを発行できない．

```
process P(chan *cha, chan *chb) {
  darea chan chi
    ...
  p = new P();
  p(cha::x, chb::chi);
}
```

2. 末尾再帰使用（A）

   rプロセスが末尾再帰を使用していれば，$recur$の度にthidが変わることはない．したがって，wプロセスの開始時にrのthidを知らせることにより，rプロセスにtrigger-contを発行することができる．しかし，新通信チャネルを獲得する$getch$ ($malloc$)命令が必要．

```
process P(chan *cha, chan *chb) {
  chan *chi = getch();
    ...
  recur(cha::x, chb::chi);
}
```

3. 末尾再帰使用（B）

   $getch$命令の代わりに，$outch$プロセスを生成して新チャネルを獲得する．通信の度にプロセスを生成しなければならない．

```
process P(chan *cha, chan *chb) {
  chan *ch1;
  int o = new outch();
  o(cp::chb, cn::&ch1);
  recur(cha::x, chb::ch1);
}
```

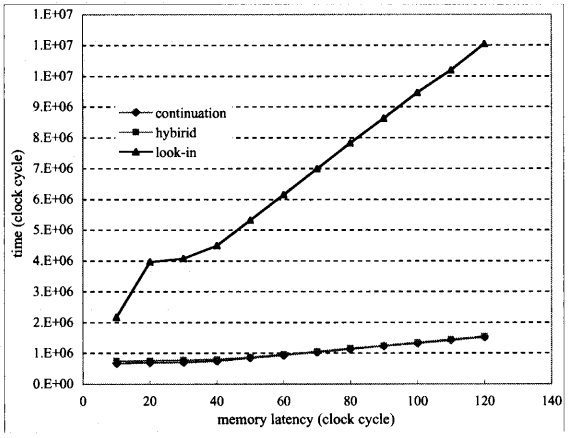

Fig. 12 Comparison between stream processing methods.

6. ストリーム処理方式の評価

たすきがけ方式（3.2節）とLook-in方式（3.4節）はbusy waitのみ（だたし，プロセッサ要素を常に占有している訳ではない）を使った方法であり，本質的にはこれらは同等とみなせる．したがって，ストリーム処理方法を大別すると，継続のみを使う方法，busy waitのみを使う方法，および両者の混合(Hybrid)の計3種類になる．本節ではこの3方式について比較評価する．

Fig. 12 に，素数生成（500個まで）のプログラムを，継続方式，Look-in方式，Hybrid方式のそれぞれにより，FUCEプロセッサシミュレータ上で実行した結果を示す．横軸にメモリアクセスレイテンシをとっており，FUCEプロセッサの動作周波数の変化に伴う性能変化を見ることができる．容易に予測できたことであるが，Look-in方式の性能悪化が著しい．これはbusy waitのために全ての

プロセスが事実上実行中であり，メモリアクセスの頻度が高いからである．その上，プロセッサ要素や待ち行列等の計算資源も無駄に消費している．そもそも，素数フィルタのパイプラインにおいて，上流のフィルタプロセスほど忙しく，下流にいくほどストリーム流量が小さくなるため，同時に有効な処理を行えるプロセスの数は実はあまり多くないのである．

一方，継続方式とHybrid方式はほぼ同性能であるが，継続方式の方がやや速く，両者の最大速度差は10%程度であった．プログラミングの観点からは，継続方式が最も実装しやすく，次いでHybrid方式であった．以上より，実行性能と実装のしやすさの両面において，継続方式が最も適しているといえる．ただし，本稿ではストリーム処理を重点的に扱ったが，一般の並行プログラミング事例の中には，資源共有問題を継続方式だけで解決することが困難な場合もある．そのような場合，Hybrid方式の採用も検討に値するといえよう．

7. む　す　び

本稿は，FUCEアーキテクチュア上での並行プログラミングの方法と言語について述べた．スレッドのfan-inカウントという，同期の要となる最小限のハードウェアサポートを活用して，ストリーム処理を簡便に記述する手法をいくつか比較検討した．

拡張IMLの並行処理記述のレベルに関し，他言語と比較すると，論理変数を利用して入出力方向を動的に定めることができるKL1等の並列論理型言語よりも低位である．しかし，IMLはKL1と同様，データ同期機構を備えており，プログラミング例からも分かるように，ストリーム処理の記述に関してはKL1なみの十分な抽象度を持っている．また，Java等によるマルチスレッドプログラミングと比べると，同期制御回りの記述の煩雑さがはるかに軽減されている．一方，プログラミングパラダイムとしてみたとき，走り切りスレッドと継続のみに基づいて，中断を起こさないようにプログラミングすることは，他のアプローチにはない斬新なスタイルであり，慣れを要するが，これにより得られる効果は大である．また，OSから応用に至るまでこのパラダイムで記述する試みは本研究が初めてである．

さらに，徹底した継続方式のプログラムはhand-shakeに基づく非同期回路の記述と類似しており，IMLとVerilog等との関連性も興味深い．今後は，OS記述等の実証を進める一方，IMLプログラミングからハードウェア記述の分野まで，研究を発展させたいと考えている．

謝　　辞

本研究は，日本学術振興会 科学研究費補助金 基盤研究(A)「細粒度マルチスレッド処理原理による並列分散処理カーネルウェアの研究」(課題番号: 15200002)の一環として行った．

参 考 文 献

1) 雨宮聡史，松崎隆哲，雨宮真人．“排他実行マルチスレッド実行モデルに基づくオンチップ・マルチプロセッサの設計”，情報処理学会 計算機アーキテクチャ研究会，2003-ARC-155，pp.51-56，(2003).
2) Herbert H.J. Hum et al. A Design Study of the EARTH Multiprocessor, In the Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'95), pp.59-68, (1995).
3) Kazunori Ueda and Takashi Chikayama., Design of the kernel language for the parallel inference machine, The Computer Journal, pp.494-500, (1990).
4) Kentaro Inenaga, Shigeru Kusakabe, Tetsuro Morimoto, and Makoto Amamiya. Hybrid Approach for Non-strict Dataflow Program on Commodity Machine, Proc. of Intl. Symp. on High Performance Computing (ISHPC'97), pp.243-254, (1997).
5) Shigeru Kusakabe et al. Implementation of a Non-strict Functional Programming Language V on a Threaded Architecture EARTH, Proc. of Intl. Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'98), p.95, (1998).
6) Shigeru Kusakabe et al. Implementing a Non-strict Functional Programming Language on a Threaded Architecture, Fourth Intl. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'99), pp.138-152, (1999).
7) Péter Kacsuk and Makoto Amamiya. A Multithreaded Implementation Concept of Prolog on Datarol-II Machine, Proc. of Intl. Symp. on High Performance Computing (ISHPC'97), pp.91-106, (1997).
8) Zsolt Németh, Hiroshi Tomiyasu, Péter Kacsuk, and Makoto Amamiya. Multithreaded LOGFLOW on KUMP/D, Proc. of Second Intl. Symp. (ISHPC'99), pp.320-327, (1999).
9) Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data Structures for Parallel Computing, Technical Report TR-87-810, Cornel University, (1987).
10) Makoto Amamiya and Rin-ichiro Taniguchi. Datarol: A Massively Parallel Architecture for Functional Language, Proc. IEEE 2nd SPDP, pp.726-735, (1990).
11) David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine, In Proc. of 4th ASPLOS, pp.164-175, (1991).