システムソフトウェアと
オペレーティング・システム 67－12
(1994. 12. 8)

並列 C++ 言語システム $\mathcal{ANUFO}$ の同期機構

川上 かおり, 佐藤 裕幸

三菱電機株式会社
情報システム研究所

〒247 神奈川県鎌倉市大船5丁目1番1号

{kaori,hiroyuki}@isl.melco.co.jp

あらまし $\mathcal{ANUFO}$ は、分散/並列処理と親和性の高いオブジェクト指向機能を備えたC++言語をベースとした並列言語及びデバッガ、通信モニタなどから構成される並列プログラミング・システムである。本報告では、$\mathcal{ANUFO}$ の同期機構に関し、その実現方式及び評価結果について詳述する。$\mathcal{ANUFO}$ の同期機構は、関数呼び出し時のものと、同期機能付き変数によるものに大別される。関数呼び出し時の同期では、呼び出し関数及びそれにより生成された子プロセスの終了検知は重み付き参照カウントを応用することにより効率良く行なっている。また、同期機能付き変数を各プロセッサで分散管理することで、管理のための通信量が削減された。

和文キーワード 並列処理、プログラミング言語、C++、同期機構、PVM

Parallel C++ Language System $\mathcal{ANUFO}$ — Synchronization Mechanism —

Kaori Kawakami, Hiroyuki Sato

Mitsubishi Electric Corporation
Computer and Information Systems Laboratory

5-1-1, Ofuna, Kamakura, Kanagawa, 247, Japan

{kaori,hiroyuki}@isl.melco.co.jp

Abstract

$\mathcal{ANUFO}$ is a parallel programming system which consists of a parallel object-oriented programming language based on C++, a debugger and a communication monitor. In this report, the implementation and some evaluation results are discussed on $\mathcal{ANUFO}$'s synchronization mechanism. $\mathcal{ANUFO}$ has two types of synchronization mechanism, one is to declare to synchronize when a function is called, and the other is to synchronize by a special variables. In the implementation for the former, the termination of the called function is efficiently detected by the weighted reference counting scheme. The creation and manegement of the variables for synchronization are distributed to each processor to prevent a communication(traffic) bottleneck.

英文 key words Parallel processing, Programming language, C++, Synchronization mechanism, PVM

1 はじめに

我々は、様々な分散 / 並列処理環境における並列処理記述を容易にすることを目的とし、分散 / 並列処理と親和性の高いオブジェクト指向機能を備えたC++ 言語をベースとした並列言語システム *ANUFO*(A NORMA-base User-Friendly Object-Oriented Language and Programming System)[1] の研究 / 開発を行なっている。

並列プログラミングにおいては、ユーザに対し、どの様な形式で通信機能と同期機能を提供するかが、並列処理記述を容易にするかどうかを大きく左右する。

ANUFO ではこのうちの同期機能を、並列に動くオブジェクトへのメンバ関数呼び出し時の同期と、同期変数と呼ばれる特別な変数を導入することにより実現しており、簡潔な記述による高度な機能の提供を可能としている。

本報告は、このような *ANUFO* の「関数呼び出し時の同期機構」及び「同期機能付き変数による同期機構」に関し、その仕様及び実装方式について詳述し、これらの機能を実現するための実行時ライブラリの性能評価を行なうものである。

2 *ANUFO* の同期機構

ANUFO の同期機構には、「関数呼び出し時の同期機構」と「同期機能付き変数による同期機構」の2つがある。前者は、並列に動くオブジェクト (以下、リモートオブジェクトと呼ぶ) へのメンバ関数呼び出し (以下、リモートメンバ関数呼び出しと呼ぶ) 時に同期をとる方法であり、メンバ関数呼び出しにおいて、呼び出された側の実行が完了するまで、呼び出し側の実行を中断することで同期をとるメカニズムである (処理同期)。

後者は、データのアクセス順序で実行順序を制御する変数 (以下、同期変数と呼ぶ) により同期をとる方法であり、同期変数に対する書き込み要求が完了するまで、読み出し要求の実行を中断することで同期をとるメカニズムである (データ同期)。

2.1 関数呼び出しによる同期

リモートメンバ関数呼び出しによる同期機構は、メンバ関数呼び出しの返り値の有無やキーワード wait を付加することにより、処理の実行順序を制御するものである。

- 返り値を使用する呼び出し
  例:
  ```
  int X = obj -> f(...);
  g(...);
  ```
  f(...) の実行が終了するまで、すなわち返り値が変数 X に代入されるまで、次のステートメント g(...) の実行に進まない。

- 返り値がない / 返り値を使用しない呼び出し
  例:
  ```
  obj -> f(...);
  g(...);
  ```
  f(...) の実行の終了を待たずに、次のステートメント g(...) の実行に進む。

- 返り値を同期変数に代入する呼び出し
  例:
  ```
  sync int *X = obj -> f(...);
  g(...);
  ```
  f(...) の実行の終了を待たずに、次のステートメント g(...) の実行に進む。

- キーワード wait を前に付けた呼び出し
  例:
  ```
  wait obj -> f(...);
  g(...);
  ```
  f(...) の「全ての実行」が終了するまで、次のステートメント g(...) の実行に進まない。ここで「全ての実行」とは、f(...) の中から、さらに呼び出されるリモートメンバ関数呼び出しの実行を意味する。

返り値による同期と、キーワード wait による同期の違いは、返り値による同期の場合、返り値を持つ関数自体の実行が終れば次の処理に進むのに対して、wait の場合は、その関数の中で呼び出されたリモートメンバ関数呼び出しの終了まで待つ点である。キーワード wait を導入して、すべての子孫呼び出しの処理の終了の検知を可能にすることにより、ユーザがプログラム中で他のプログラムのスケジュールを行なうような、メタなプログラミングが可能となる。

2.2 同期変数による同期

ANUFO における同期変数は、sync 変数、stream 変数の2つである。

2.2.1 sync 変数

write once/multiple read 型の同期変数を、sync 変数と呼ぶ。変数には未定義 / 定義の2つの状態があり、未定義の状態での読み出しは書き込みが起きる (これにより定義状態になる) まで中断し、定義状態での書き込みはエラーになる。それ以降の読み出しに対しては、常に同じ値を返す。sync 変数はいわゆる I-Structures[3] であるが、本言語では構造体ではなく、int などのアトミックなデー

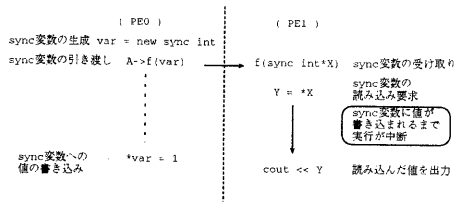

図 1: sync 変数の書き込み / 読み出し

タにもこの機能を取り入れている部分が大きく異なる。

sync 変数は、ある 1 人の生産者がデータを生成しそのデータを消費者が必要とする場合に、消費者では単に読み出すという記述だけで、データの生成が完了するまで待つことができる。

```
《sync 変数の記述例 1 (書き込み / 読み出し)》
class obj {
int Y;
public :
  void f(sync int* X){//sync 変数の受けとり
    Y = *X;       //var の読み出し要求が発生
    cout << Y;
        :
  }
};
main()
{
  sync int *var;
  remote obj *A;
  A = new remote obj@PE1
        //PE1 に並列オブジェクト obj を生成
  var = new sync int; //sync 変数 var を生成
  A->f(var);
       //sync 変数 var をオブジェクト A に渡す
     :
  *var = 1; //var に 1 が書き込まれる
}
```

図 1は記述例 1 に基づく、sync 変数の書き込み / 読み出しの手順である。

1. sync 変数 var を sync int *var で宣言し、var = new sync int で生成する。この時の sync 変数の生成はデフォルトプロセッサ 0(PE0) 上になされる。

2. リモートオブジェクト obj が PE1 上に生成され、Y = *X で sync 変数 X(すなわち var) への読み出し要求が発生する。その時点でまだ PE0 で var が書き込まれていなければ、PE1 の読み出し要求は中断する。

3. PE0 で *var = 1 により sync 変数 var に 1 が書き込まれると、PE1 の読み出し要求が動作しはじめ、X に値 1 が読み出される。

これは共有した 1 つの sync 変数を用いる場合である。一方、これに対し並列実行される複数のプロセスにおいて、それぞれのプロセスで生成された sync 変数が、処理の途中で同一の同期タイミングであることが判明し、それらの sync 変数を同一の変数として扱いたくなる場合がある。このような場合のために、*ANUFO* では、マージと呼ばれる、生成済みの 2 つの sync 変数を同一の変数として取り扱えるように変換する操作を提供している。ある 2 つの sync 変数に対し、マージ操作を行なうと、一方の変数に書き込んだデータを、もう一方の変数から読み出すことができる。但し、2 変数が共に定義状態の場合は、たとえ同じ値を持っていてもマージできない。

```
《sync 変数の記述例 2 (マージ)》
sync int *W, *X, *Y, *Z;
            //W,X,Y,Z は sync 変数
int V;
   :        //sync 変数の生成など
*W = 1;     //W に 1 を書き込む
merge(W,X); //W(定義状態) × X(未定義状態)
merge(Y,Z); //Y(未定義状態) × Z(未定義状態)
merge(X,Z); //X(定義状態) × Z(Y を参照)
V = *Y;     //V = 1 : Y を V に読み込む
```

記述例 2 は、sync 変数のマージの記述例である。

2 つの sync 変数 X と Y のマージは merge(X,Y) と記述する。

W に 1 を書き込んで、さらに W と X、Y と Z、X と Z をマージすることにより、W,X,Y,Z のどの変数からも 1 を読み出すことができる。

2.2.2 stream 変数

データを複数回書き込みでき、書き込まれた順に FIFO でデータが読める同期変数を、stream 変数と呼ぶ。stream 変数に対する読み出しは、データがない場合に中断する。

stream 変数と sync 変数の機能の違いは、sync 変数への書き込みは 1 度だけで、その後は何度読んでも同じ値が読み出されるのに対して、stream 変数に対しては読み出しと同じ回数だけの書き込みが行なえ、(同じ値が書き込まれない限り) 全て異なる値が読み出される点である。

sync 変数は値を一度しか代入できなかったので、同期を取ることが中心であり、変数値そのものにあまり意味がなかったが、stream 変数はデータを何度も代入できるので、同期機構を簡便に記述できるだけでなく、データ通信にも利用できる。

stream 変数に対してもマージの機能を提供しており、sync 変数の場合と同様に、一旦マージされた 2 つの stream 変数はあたかも同一の変数のように振舞う。stream 変数同士のマージは、2 変数が未定義 / 定義状態すべての場合に可能である。

```
〈stream 変数の記述例〉
stream int *X,*Y; // stream 変数 X,Y を宣言
int Z;
   :              // stream 変数の生成など
*X = 11, *X = 12; // X に 11,12 を書き込む
                  // X=[11,12]
*Y = 21, *Y = 22; // Y に 21,22 を書き込む
                  // Y=[21,22]
merge(X,Y); // X と Y をマージ
            // X=Y=[11,12,21,22]
*X = 13;    // X に 13 を書き込む
            // X=Y=[11,12,21,22,13]
*Y = 23;    // Y に 23 を書き込む
            // X=Y=[11,12,21,22,13,23]
Z = *X;     // X を Z に読み込む
            // Z=11,X=Y=[12,21,22,13,23]
Z = *Y;     // Y を Z に読み込む
            // Z=12,X=Y=[21,22,13,23]
```

stream 変数 X を `stream int *X` で宣言し、`X = new stream int` で生成する。stream 変数への書き込み / 読み出し / マージのシンタックスは sync 変数と同じである。

コメント文の中では、stream 変数が保持する値の集合を [···] で示している。

共に stream 変数である X と Y をマージすることにより、結合集合が生成される。X,Y どちらに対する操作 (読み / 書き) も、結合集合に対して行なわれる。一度、読み込まれた変数値は、結合集合から取り出される。したがって、次の読み出しでは、集合上の異なる変数値が読める。

3 関数呼び出しによる同期の実装方式

リモートメンバ関数呼び出し時に同期をとる方法には、メンバ関数にキーワード wait をつける方法と同期変数ではない変数への返り値を受けとる方法の 2 つがある。*ANUFO* では、このうちキーワード wait 付きリモートメンバ関数呼び出しの終了検出を WTC(Weighted Throw Counting) 方式 [4] と呼ばれる効率的な手法によって実現している。WTC 方式では、子プロセスの生成時に “weight” と呼ばれる値を与え、各プロセスの処理が終了した時に、プロセス木のルートとなったプロセスで “weight” を回収することにより、終了検知を実現している。

ANUFO では、リモートオブジェクトのメンバ関数の実行が、WTC の子プロセスに相当する。

```
[ プログラム 1 ]
class obj{
public:
  remote obj *obj21, *obj22, *obj31, *obj32;
  void f(int PE2, int PE3) {
    obj21 = new remote obj@PE2;
    obj22 = new remote obj@PE2;
    obj31 = new remote obj@PE3;
    obj32 = new remote obj@PE3;
    obj21->f1(); obj22->f1();
    obj31->f1(); obj32->f1();
      :
  }
  void f1(){...}
};
```

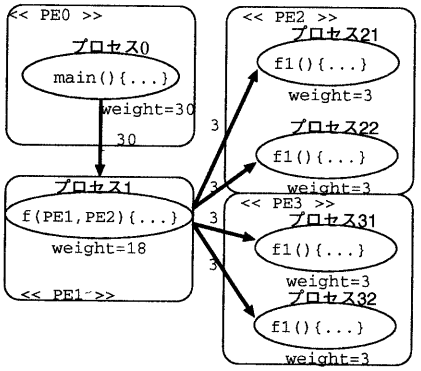

図 2: weight の分配

```
void g(){...}
main()
{
  int PE1,PE2,PE3;
  remote obj *obj1;
    :
  obj1 = new remote obj@PE1;
  wait obj1 -> f(PE2,PE3);
  g();
}
```

具体的な実装方法の例として、プログラム 1 の実行を考える。

図 2に weight の分配方法について示す。

PE1,PE2,PE3 はリモートオブジェクトが生成されるプロセッサである。main() からの、リモートメンバ関数呼び出し f(PE2,PE3) の実行により、全ての子プロセス (プロセス 1, プロセス 21, プロセス 22, プロセス 31, プロセス 32) に対し、weight が分配される。

- ルートであるプロセス 0 からプロセス 1 へ、weight が仮に 30 渡されたとする。
- 一般的に、親プロセスから子プロセスに対して、親プロセスの weight の定数分の 1 の値が weight として与えられる。プロセス 1 では 4 つのプロセスに対して、3(= 30/10) が weight として与えられ、これにより、プロセス 1 の weight は $30 - 3 \times 4 = 18$ となる。
- weight がこれ以上分割不可能な値 ((例) 整数値 1) であったら、ルートプロセスに追加の weight を要求する。ルートプロセスから新たに weight が与えられ、それにともないルートプロセスの weight 値も更新される。

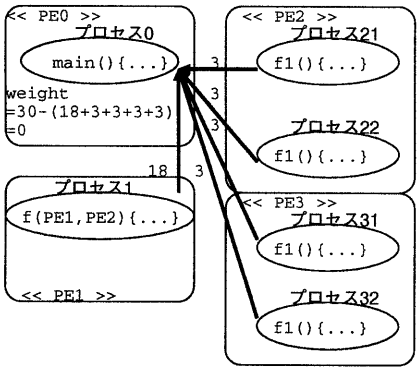

図 3: weight の回収

次に、図 3で weight の回収の手順を示す。

- 各プロセスの処理が終了したら、保持している weight をルートプロセス (例ではプロセス 0) に返す。
- 受けとった weight をルートプロセスが保持する weight から引いて行くと、最終的に $30-(18+3+3+3+3)=0$ となり、全てのプロセスの終了を検知する。

各プロセスは他のプロセスに依存しないで独立に処理を終了できる。これは各プロセスの処理が終了した時に、そのプロセスを直接生成した親プロセスに weight を返すのではなく、ルートプロセスに weight を返すためである。

4 同期変数の実装方式

同期変数はユーザプログラムを実行するタスク (以下、ユーザタスクと呼ぶ) と、同期変数を管理するタスク (以下、サーバタスクと呼ぶ) の 2 つのタスクにオブジェクトとして生成される。

4.1 同期変数処理部の構成

同期変数処理部のシステム構成を図 4に示す。

サーバタスクは、同期変数を管理するタスクであり、各プロセッサに 1 つずつ存在し、ユーザプログラムの中で同期変数へのアクセスが発生すると、このサーバタスクにメッセージが送られる。各サーバタスクは、そのプロセッサで生成されたすべての同期変数を管理し、そのための管理テーブルを 1 つ持つ。sync 変数、stream 変数、global

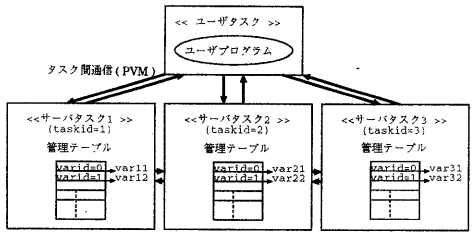

図 4: 同期変数処理部の構成

変数[1] の 3 種類の同期 / 通信変数は、同一の管理テーブル上で管理される。

このように、「管理しているサーバタスクの ID (taskid と呼ぶ)」と「管理テーブル上での個々の変数の ID (varid)」の 2 つの ID により、同期 / 通信変数は一意に表現される。

サーバタスクがシステム全体に 1 つの場合でも、同期変数の管理は可能であるが、サーバタスクを各プロセッサごとに置くことにより、管理のための通信量の観点で効率が良くなる。例えば、変数を生成したプロセッサと、変数にアクセスしようとしているプロセッサが物理的に近くに存在し、サーバタスクが 1 つの場合にそのサーバタスクを実行するプロセッサがそれらから離れている場合は、明らかに通信効率が悪くなる。

しかし、サーバの複数化によって、逆に通信量が増えてしまう場合も考えられる。異なるプロセッサ上に生成された複数の同期変数間のマージは、システム全体に 1 つのサーバであれば、サーバ内のローカルなデータ通信ですむところが、各変数を複数の異なるサーバで別々に管理している場合、異なるプロセッサに存在するサーバ間のデータ通信が発生してしまう。

この 2 つのトレードオフに着目した評価については後述する。

4.2 同期変数への操作

ANUFO では、同期変数 (sync 変数、stream 変数) を一つのオブジェクトとして実現している。同期変数オブジェクトは、ユーザタスクとサーバタスクに作られ、それぞれ、スレーブオブジェクト、サーバオブジェクトと呼ぶ。スレーブオブジェクトでは、その同期変数を管理しているサーバタスクの ID と変数の ID を保持することで、同期変

[1] すべての並列実行単位から共有される変数をいう。

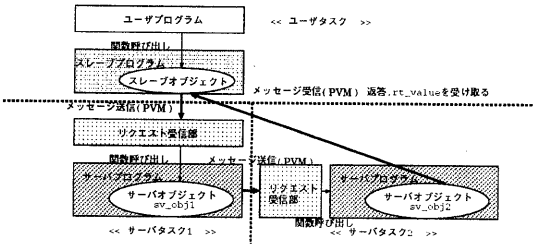

図 5: 操作の流れ

数への操作要求をサーバタスクに送っている。一方サーバオブジェクトは、同期変数の実体(変数の値)を保持し、サーバタスクの管理テーブルに格納される。

同期変数へは、生成、書き込み、読み出し、マージの4つの操作を行なえる。

あるユーザタスク上の同期変数に対する操作(図5参照)は、スレーブオブジェクトから、アクセスしたい同期変数を管理するサーバタスクに操作要求を送る。メッセージを受けとったサーバタスクは、保持する管理テーブルの該当するサーバオブジェクト(sv_obj1,varid で指定)のメンバ関数を呼び出す。メンバ関数呼び出しにより、サーバオブジェクト(sv_obj1)では、同期変数の変数値などのメンバ変数への操作を行なう。その結果(正常に終了したかどうかを示す返答、または返り値(rt_value))は、メッセージの要求元スレーブオブジェクトに返される。

マージなどの2変数にまたがる操作により、異なるサーバタスクが管理する同期変数への操作が生じると、該当するサーバオブジェクト(sv_obj2)に操作要求が転送[2]されるが、その結果は、最終的に操作を行なったサーバオブジェクト(sv_obj2)から、メッセージの要求元スレーブオブジェクトに直接返される。

5 評価

以上の仕様/実装方式に基づき、現在、*ANUFO* の第0版が数台の UNIX ワークステーションをイーサネットで接続した環境で動作している。

そこで、同期変数の以下の仕様/実装方式の有効性を示すための評価を行なった。評価環境は、sparc station10 を2～4台接続し、通信ライブラリ PVM[2] は3.3版を用いた。

5.1 複数サーバによる同期変数の管理

ANUFO では、各同期変数を生成したプロセッサごとに同期変数を管理するサーバが存在する実装方式になっている。そこで、各プロセッサで生成されたすべての同期変数を単一のサーバが管理する実装方式との実行時間の比較を行なった。

ここで、二つの実装方式の違いを明確にするために、単一サーバ版では、

- サーバが存在するプロセッサのタスクからは、同期変数にアクセスしない。
- 同期変数の書き込み/読み出しは変数を生成したプロセッサ上のタスクから行なう。

などの制約を付けた。

ANUFO の実装方式(複数サーバ版と呼ぶ)と単一サーバ版との実行時の相違点で、プログラムに依存しないものには、同期変数の生成時に、複数サーバ版ではプロセッサ間に渡らずにローカルに行なわれるのに対して、単一サーバ版ではプロセッサに跨る通信を伴って行なわれる点がある。
さらにここで用いた測定プログラムでは、同期変数の読み出し/書き込みに関しても同様に、複数サーバ版ではローカルに、単一サーバ版ではプロセッサに跨る通信を伴って行なわれる。
この2つの観点では、複数サーバ版の方が、単一サーバ版よりも実行時間が速くなることが予想される。

逆に、マージによるデータ/待ちタスクの通信が、単一サーバ版ではローカルに行なわれるが、複数サーバ版ではプロセッサ間に跨って行なわれる。これより、マージに関しては、複数サーバ版の方が実行時間が遅くなることが予想される。

さらに、stream 変数のマージの場合、sync 変数の場合よりも1回のマージで通信されるデータ数が多い分、複数サーバ版でのプロセッサ間通信が多く発生するため、sync 変数の場合よりも複数サーバ版の方が実行時間が遅くなり、実行時間の差が顕著になることが予想される。

これらを総合すると、複数サーバ版では、単一サーバ版に比べて、同期変数の生成プロセッサにサーバが存在することによって、変数アクセス時の通信量が少ない分、複数サーバに跨るマージの際にデータ通信が発生する事とのトレードオフが問題となる。

[2]必ずしも sv_obj1 に送られてきたメッセージと同じメッセージではない。

表 1: 単一サーバと複数サーバでの実行時間の比較 (ミリ秒) - 1sync 変数 - (() 内は比率を示す)

単一サーバ	26.5(1.43)
複数サーバ	18.6(1.00)

表 2: 単一サーバと複数サーバでの実行時間の比較 (ミリ秒) - 2sync 変数 - (() 内は比率を示す)

	生成 / 書き込み / マージ	マージ
単一サーバ	38.2(1.30)	7.14(0.92)
複数サーバ	29.0(1.00)	7.72(1.00)

1sync 変数の生成 / 書き込み / 読み出し 1 つの sync 変数の生成 / 書き込み / 読み出しの実行時間を測定した (表 1)。複数サーバ版の方が、単一サーバの場合より約 1.43 倍速い。これは、複数サーバ版の方が、プロセッサ間通信が少ないためと思われる。

異なるプロセッサ上の 2sync 変数のマージ 次に、異なるプロセッサ上で生成された 2sync 変数のマージ処理の時間を測定した (表 2)。実際のプログラムにおいて、マージ時にデータ通信がおきるマージは必ず生成 / 書き込みを伴う処理であることから、生成 / 書き込み / マージを行なう測定プログラムを用いて、マージ処理の評価を行なった。複数サーバ版の方が、1.3 倍弱速くなっており、前記の 1sync 変数に対する生成 / 書き込み / 読み出し時間の比較結果 (1.43 倍) と比べて、大きくはかわっていない。これより、複数サーバ版では、マージ処理の時に、プロセッサ間通信が発生してしまうことのデメリットは、あまり実行時間全体に影響ないことがわかった。

さらに、マージのみの実行時間の比較を行なうために、上記のプログラムと同じものからマージ処理を除いた (生成 / 書き込みのみの) プログラムを用いて測定し、その差分からマージの実行時間を求めた (表 2)。マージの実行時間は、複数サーバ版の方が、単一サーバ版の 1/0.92=1.08 倍遅くなる程度であり、ほぼ無視できる。

異なるプロセッサ上の 2stream 変数のマージ 以上の測定結果は単一データの場合なので、次に stream 変数を使って、マージ時に複数データの通信を行なうマージ処理を評価した。(表 3)。sync 変数と同様に、2 つの stream 変数を異なるプロセッサで生成し、一方の stream 変数に複数データの書き込みをしてから、マージを行なうプログラムの実行時間を測定した。sync 変数と同様に、複数サーバ版の方が、1.30 倍 ~ 2.22 倍速くなっているので、データ数にかかわらず、マージ時の複数サーバによるオーバヘッドはほとんどないといえる。さらに、sync 変数と同様な方法でマージのみの時間を求めたところ (表 4)、1 ~ 10,000 整数データ (4 ~ 40,000byte) において、複数サーバ版と単一サーバ版との差は誤差程度 (2 ~ 70[msec]) であった[3]。

表 3: 単一サーバと複数サーバでの実行時間の比較 (ミリ秒) - 2stream 変数 - (() 内は比率を示す)

	生成 / 書き込み / マージ		
データ数	1	100	10,000
単一サーバ	38.0(1.30)	713(2.07)	69073(2.22)
複数サーバ	29.7(1.00)	345(1.00)	31100(1.00)

表 4: 単一サーバと複数サーバでの実行時間の比較 (ミリ秒) - 2stream 変数 - (() 内は比率を示す)

	マージ		
データ数	1	100	10,000
単一サーバ	5 ~ 13	15 ~ 58	484 ~ 2070
複数サーバ	1 ~ 8.6	9 ~ 24	227 ~ 314

5.2 stream 変数によるデータ通信

データの生産者 (generator) と消費者 (consumer) だけから成る 1 対 1 のデータ通信は、リモートメンバ関数呼び出しを用いても比較的容易に記述可能である。そこで、1 対 1 通信を stream 通信を用いた場合と、リモートメンバ関数呼び出しで行なった (stream 変数を用いなかった) 場合と比較する (表 5)。

結果は stream 変数を使った場合の方が 1.5 倍程度遅かった。これは、stream 変数の場合、stream 変数の生成 / 書き込み / 読み出しの際に、サーバを経由したタスク間通信がおきるためと思われる。

この測定では、generator と consumer がそれぞれ 1 つずつ、つまり 1:1 の通信に対して評価を行なっているが、そもそも stream 変数のような同期

[3] この測定はマージ / 生成 / 書き込みを行なうプログラムと、生成 / 書き込みのみのプログラムとの差分をマージの時間としているので、生成 / 書き込み時のネットワークの状況によってかなり測定結果がふらついてしまい、複数サーバ版の方が速い場合も多くみられた。

表 5: stream 変数を使用 / 使用しない場合の実行時間の比較 (ミリ秒) (() 内は比率を示す)

stream 変数を用いた場合	9.16(1.52)
stream 変数を用いなかった場合	6.01(1.00)

変数は、データの準備ができたら処理を開始するということを簡便に記述するために導入した機能である。generator 又は consumer が 1 つの場合であれば、stream 変数を使用しないでリモートメンバ関数呼び出しを用いても記述できるが、やはり、記述性の面で同期変数のメリットは多い。更に、多対多の通信を stream 変数を使用せずに記述するには、データを集中管理するタスクが必要となり、結局、同期変数と同じような処理をしなければならなくなり、その場合の効率は同期変数と変わらないであろう。

現在の stream 変数は、多対多の通信も可能なように、汎用的な仕様になっている。従って、1 対 1 の通信を効率良く処理するような新しい同期変数の導入が、今後の改良項目の 1 つとして挙げられる。

6 おわりに

以上の方式により同期機構を実装 / 評価した。

関数呼び出し時の同期機構では、並列に動くプロセスの処理の終了検知に、WTC 方式に基づく方法を導入したことにより、効率の良い終了検知が可能になった。

同期変数に関しては、「各プロセッサに変数オブジェクトを管理するサーバを設ける。」などの通信効率を考慮したシステム設計 / 実装により、実行時間 / 通信量が削減された。

また、同期変数を同期変数クラスのオブジェクトとして、C++ の " 継承 / テンプレート / 関数の上書き " などの機能を用いて実現したことにより、同期変数ライブラリの効率の良い実装が可能になった。

今後は、stream 変数の 1:1 通信の仕様の検討等を行ない実装 / 評価する予定である。

参考文献

[1] 佐藤裕幸, 川上かおり: 並列 C++ 言語システム $\mathcal{ANUFO}$ の並列処理機構, 電子情報通信学会コンピュータ研究会 (CPSY) 技術研究報告, 1994.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam: "PVM 3.0 User's Guide and Reference Manual", Engineering Physics and Mathematics Division, Mathematical Sciences Section, Oak Ridge Natinal Laboratory, Oak Ridge, Tennessee, Feb. 1993.

[3] ARVIND, R. S. NIKHIL, K. K. PINGALI: "I-Strucutres : Data Structures for Parallel Computing", ACM Transactions on Programming Languages and Systems, Vol.11, No.4, pp.598-632, 1989.

[4] K. Rokusawa, N. Ichiyoshi, T. Chikayama, H. Nakashima: "An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems", Proc. ICPP: International Conference on Parallel Processing, Vol.I, pp.18-22 (1988).