# マスタ–ワーカモデルから見たRisa/Asir並列プログラムとCell用SPEライブラリ並列プログラムの比較

白石 啓一[*]　　今井 慈郎[†]

## A discussion of special-purpose parallel programming based on master-worker paradigm – Comparison of Risa/Asir program and Cell B.E. one with SPE library –

Keiichi SHIRAISHI　　Yoshiro IMAI

**Abstract**

This paper describes special-purpose parallel programming based on a master-worker paradigm. Such programming can be realized and executed on multi-core processor computers. We have compared two types of parallel programs. One is a program for a computer algebra system called Risa/Asir, which is executed on a personal computer with 4 cores processor. Another is for Cell B.E. with SPE library on PLAYSTATION3. Though a real experiment, it has been confirmed that the former is suitable for symbolic and/or algebraic computation while the latter is efficient for numerical computation.

*Key Words:* Master-Worker Paradigm, Multi-Core Programming, Parallel Programming

## 1　はじめに

近年，マイクロプロセッサの急速な進歩やメモリ，磁気ディスク容量の大幅な向上による，計算機(ワークステーション(WS)，パーソナルコンピュータ(PC))自体のコモディティ化(部品化)，計算機を接続するネットワークの高速化などにより，クラスタコンピューティングが広まっている．マイクロプロセッサの発展も単なる速度向上ではなく，マルチコア化へ進み，1プロセッサが同時に複数の処理を実行できるようになった．そのため，クラスタコンピューティングによる並列処理とクラスタを構成する計算機のプロセッサ内での並列処理をうまく組み合わせる必要がある．

マルチコアプロセッサを搭載した計算機クラスタでの並列処理プログラム作成を目指し，本稿では，1台の計算機での並列処理プログラム実行に焦点を当てた．4コアのプロセッサを搭載するPC上で動作する数式処理システムRisa/Asirを用いた並列計算，6コアのプロセッサを搭載するCell B. E.を用いた並列計算を取り上げ，マスタ–ワーカモデルの視点から，両者を比較する．

## 2　並列処理について

並列計算機には，複数のスカラ計算機を同時に使用する計算機クラスタ，多数のデータをまとめてベクトルデータとして計算するベクトル計算機などが

[*]香川高等専門学校詫間キャンパス　通信ネットワーク工学科
[†]香川大学　工学部

ある．次に示すアムダールの法則 [1] は，ベクトル計算機について述べているが，計算機クラスタも同様で，並列演算可能な部分の割合 (アムダールの法則ではベクトル化率) を上げないと高い総合性能向上率は望めない．

**アムダールの法則**　あるプログラムをスカラ計算機で実行した場合の実行時間を $T_S$，ベクトル計算機で実行した場合の実行時間を $T_V$，このプログラムのベクトル演算が可能な部分の割合—ベクトル化率を $\alpha$，ベクトル演算による性能向上比を $V$ とすると，総合性能向上率 $P$ は

$$P = \frac{T_S}{T_V} = \frac{1}{(1-\alpha)+\alpha/V} \qquad (1)$$

で表される．

**マスタ–ワーカモデル**　マスタ–ワーカモデルは並列処理のプログラミングモデルの一つで，処理全体を制御するマスタと処理の一部 (仕事) を担うワーカが協調して動作するモデルである．図 1 はワーカ 3 個の場合を示しており，

1. マスタが各ワーカへ仕事を割り当てる
2. 各ワーカは仕事を行い，結果をマスタへ返す
3. マスタは結果を受け取り，全ての仕事を終えてなければ 1 へ戻る

という手順で処理が進む．

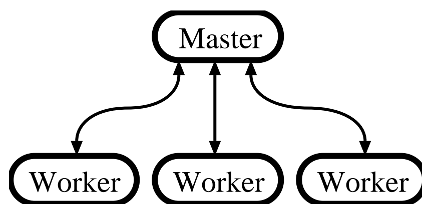


図 1　マスタ–ワーカモデル

マスタ–ワーカモデルにおいて，ワーカへ割り当てる仕事の処理時間が均等であれば，式 (1) どおりの総合性能向上率が望める．仕事の処理時間が均等でない場合，どの仕事を先に実行するか (どのようにスケジュールを作るか) で，全体の処理時間が変化する．この問題に関して，次に示すスケジュール方法が知られている．

**故意の休止を含まないスケジュール**　最適なスケジュールを作るための (徹底した試行錯誤よりも短い) アルゴリズムは，未だ知られていないが，故意の休止を含まないスケジュールを立てることで比較的良いスケジュールを作成できることが知られている．故意の休止を含まないスケジュールでは，実行可能なジョブがないときのみ，プロセッサを休ませる．“故意の休止を含まないスケジュール” がどの程度良いものかを示す次の定理がある [2]．

**定理 1**
与えられたジョブの集合に対して，ジョブが故意の休止時間を含まないあるスケジュールに従って実行されるときの全経過時間を $\omega$ で表し，可能な最小全経過時間を $\omega_0$ で表す．このとき，

$$\frac{w}{w_0} \leq 2 - \frac{1}{n}$$

となる．ここで， $n$ はその計算システムにおけるプロセッサの台数である．さらに，この上界は最良である．

つまり，仕事が発生した順にワーカへ割り当てれば，最適スケジュールによる最小処理時間の高々2 倍の時間で処理を終えることができる．ただし，本節では，マスタ–ワーカ間の通信や同期などのオーバーヘッドを考慮していないので，その点での工夫の余地がある．

## 3　Risa/Asir 並列プログラムと Cell 用 SPE ライブラリ並列プログラムの比較

数式処理システム Risa/Asir 用の並列プログラムは，次に示す順に処理を進める必要がある．

1. ワーカとなる asir プロセスを起動し，ワーカを割り当てる (`ox_launch()` の引数を変えることで，同一計算機でも他の計算機でも起動できる)
2. ワーカプログラムを各ワーカへロードする (`ox_rpc()`, `load()`)
3. ワーカプログラムを各ワーカで実行する (`ox_rpc()`)
4. ワーカプログラムの実行終了を待ち，各ワーカから結果を取得する (`ox_pop_local()`)
5. 全ての仕事を終えてなければ，3 へ戻る

マルチコアプロセッサを使った場合，各コアへのプロセス割当は OS の負荷分散機能により自動的に行われる．マルチコアプロセッサを持つ計算機でも，計算機クラスタでも `ox_launch()` の引数が異なるだけで，同じインターフェースでプログラムを書くことができる．Risa/Asir では，ワーカを明示的に破棄する必要はない．`ox_pop_local()` は，ワーカ

プログラムの終了待ちと処理結果取得の機能を持っているので，マスタはワーカプログラムが終了するまで待ち続けることになる．ワーカへ割り当てる仕事の処理時間が均等である場合，ほとんど問題は起こらないが，均等でない場合，空いたワーカに次の仕事を割り当てれば，効率が良くなる．その場合，`ox_push_cmd()`, `ox_select()` を用いて，ワーカプログラムが実行終了したものを調べ，終了したものから `ox_get()` により処理結果を取得する．

SPE ライブラリバージョン 2 を用いた Cell B.E. 用の並列プログラムは，次に示す順に処理を進める必要がある [3]．

1. ワーカプログラムをロードする (`spe_image_open()`, `spe_program_load()`)
2. 各 SPE へワーカを割り当てる (1 のワーカプログラムが必要, `spe_context_create()`)
3. `pthread_create()` により，スレッドを生成し，スレッド内で各 SPE へワーカプログラムの実行を指示する (`pthread_create()`, `spe_context_run()`)
4. 各スレッドは，ワーカプログラムが終了したときに，終了する
5. `pthread_join()` により各スレッドの終了を待つ
6. ワーカを破棄する (`spe_context_destroy()`)
7. ワーカプログラムを閉じる (`spe_image_close()`)

ワーカプログラムの実行を指示する `spe_context_run()` は，ワーカプログラム実行終了まで戻り値を返さない関数なので，POSIX thread でスレッドを生成し，他のワーカを有効利用する必要がある．Cell では，データの受渡しを SPE が主体となり DMA 転送などにより行う．SPE で動作するワーカプログラムは，DMA 転送によりデータをメインメモリからローカルストアへ転送し，処理し，再度 DMA 転送によりデータをメインメモリへ転送する．

Risa/Asir と SPE ライブラリの関数名の比較表を表 1 に示す．マスタ–ワーカモデルを実現するために必要な関数が揃っていることが分かる．処理手順に示したとおり，一部の機能と実行順序が両者で異なっている．

## 4 円周率計算

円周率 $\pi$ の値は，

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \qquad (2)$$

であると知られている．式 (2) を数値積分することで計算できる [4]．

$$\pi \approx \frac{1}{N} \sum_{i=0}^{N-1} \frac{4}{1+((i+0.5)/N)^2} \qquad (3)$$

ここで，$N$ は，式 (2) の積分区間 $[0, 1]$ の分割数である．$N$ を大きくすることで精度良く $\pi$ の値を求めることができる．この計算は，$i$ に関するループで記述でき，このループをワーカへ割り当てることができる．つまり，アムダールの法則におけるベクトル化率が非常に高く，総合性能向上率が高くなることが期待できる．

Risa/Asir 用のプログラムを図 2 に示す．Risa/Asir はインタラクティブなソフトウェアであり，次に示す手順により，3 節で示した並列処理を行うことができる．

1. `asir` コマンドにより Risa/Asir を起動する (マスタになる)
2. `load("cpi2.asir");` によりプログラムをロードする
3. `ID=[ox_launch(), ox_launch(), ox_launch(), ox_launch()];` により 4 個のワーカを起動する
4. `rpc_cpi2_init(ID);` によりワーカプログラムを各ワーカへロードする
5. `cpi(ID, 50000000);` により，$N = 50000000$ のときの並列計算を行う

SPE ライブラリ用のプログラムを図 3, 4, 5 に示す．図 3 は，PPE-SPE 間で受け渡すデータのデータ構造を定義したヘッダファイルである．図 4 がマスタプログラムであり，3 節で示した手順に従い，処理している．図 5 がワーカプログラムである．ワーカプログラムは，DMA 転送によってパラメータと計算用データを取得し，計算し，再度 DMA 転送によって計算結果をマスタへ送る処理をしている．

図 6 にコンパイル用の Makefile を示す．マスタプログラムは POSIX thread ライブラリ，SPE ライブラリをリンクする必要がある．ワーカプログラムは，専用のコンパイラでコンパイルする必要がある．

表 1 Risa/Asir と SPE ライブラリの比較

| | Risa/Asir | SPE ライブラリ |
|---|---|---|
| ワーカ割当 | ox_launch() | spe_context_create() |
| ワーカプログラムロード | ox_rpc(), load() | spe_image_open()<br>spe_program_load() |
| ワーカプログラム実行 | ox_rpc() | pthread_create()*1<br>spe_context_run() |
| ワーカプログラム終了待ち | ox_pop_local()<br>or<br>ox_push_cmd(), ox_select() | pthread_join()*1 |
| ワーカ処理結果取得 | ox_pop_local()<br>or<br>ox_get() | spu_mfcdma64(), spu_writech(),<br>spu_mfcstat()*2 |
| ワーカ破棄 | ox_shutdown() | spe_context_destroy() |
| ワーカプログラムクローズ | | spe_image_close() |

*1 POSIX thread ライブラリの関数である．
*2 DMA 転送による結果取得である．マスタからワーカへのデータ転送にも利用する．

実行環境を表 2 に示す．PLAYSTATION3 の Cell B.E. は SPE を 7 個持つが，SPE ライブラリで利用できるのは 6 個である．並列計算による速度向上を確認するため，$N = 50000000$ とした．PC 上で Risa/Asir を用い，ワーカ数を 1 から 4 まで変化させたときの計算時間を表 3 に，PLAYSTATION3 上で SPE ライブラリを用い，ワーカ数を 1 から 6 まで変化させたときの計算時間を表 4 に示す．表 3, 4 により，速度向上比がほぼワーカ数に比例して上がっていることが分かる．

```
typedef struct {
  unsigned long long  ea_in;
  unsigned long long  ea_out;
  unsigned int        size;
  int                 pad[3];
} pi_params_t;  //8+8+4+4*3=32bytes

typedef struct {
  int  start, end;
  int  pad[2];
  double  width, s;
} pi_data_t;  //4*4+8*2=32bytes
```

図 3 SPE ライブラリを用いた $\pi$ 計算プログラム (pi.h)

図 7 は，Risa/Asir による並列計算中の top コマンド実行結果の抜粋である．図 7 の 10–13 行目の C 欄 (プロセスを実行しているプロセッサナンバー) が 0–3 になっていることから，ワーカである asir プロセスが各コアに割り当てられていることが分かる．

## 5 おわりに

マスタ–ワーカモデルを用いた，数式処理システム Risa/Asir 並列プログラムと Cell B.E. 用 SPE ライブラリ並列プログラムを比較した．Risa/Asir では，ox_launch() の引数を適切に使い分けることで，マルチコアプロセッサを持つ計算機上でも計算機クラスタ上でも同じプログラムを使うことができる．SPE ライブラリは，Cell が持つ SPE の利用に特化しているので，複数の SPE をワーカとして同時に動かすために POSIX thread ライブラリを必要とし，PLAYSTATION3 クラスタを組むと TCP/IP を使って通信する必要がある．

```
# PPU
PPU-GCC = gcc
PPU-CFLAGS = -lpthread -lspe2 -Wall -Werror
PPU-SRC = pi-ppe.c
PPU-OUT = pi-ppe

# SPU
SPU-GCC = spu-gcc
SPU-CFLAGS = -Wall -Werror
SPU-SRC = pi-spe.c
SPU-OUT = $(SPU-SRC:.c=.elf)

all:
$(PPU-GCC) $(PPU-SRC) -o $(PPU-OUT) $(PPU-CFLAGS)
$(SPU-GCC) $(SPU-SRC) -o $(SPU-OUT) $(SPU-CFLAGS)
```

図 6 SPE ライブラリを用いたプログラムのための Makefile

Risa/Asir は数値計算のためのソフトウェアではないので，Cell を含む数値計算と計算時間を比較することに無理がある．やはり，数式処理らしい問題を解くために利用すべきである．マスタ–ワーカモデルでのプログラム作成には十分な機能が備わっており，数式の並列処理を進められる．

現在，Cell 用の並列ライブラリには，SPE ライブラリよりも抽象化が進んだ MARS などがある．今

表 2 実行環境

| | PC | PLAYSTATION3 |
|---|---|---|
| Processor | Xeon X3210 2.13GHz | Cell B.E. 3.2GHz<br>PPE + SPE × 7<br>SPE Local Store 256KB<br>EIB 307.2GB/s |
| Memory | 3GB | 256MB |
| OS | FreeBSD/amd64 7.1-RELEASE | Yellow Dog Linux 6.2 |
| CAS, Library | Risa/Asir 20070806 | libspe-2.2.80-132 |

表 3 計算時間 (Risa/Asir)

| ワーカ数 (個) | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 計算時間 (s) | 37.902 | 19.627 | 13.787 | 11.056 |
| 速度向上比 ∗ | 1 | 1.931 | 2.749 | 3.428 |

∗ ワーカ 1 個での計算時間/計算時間

後は，MARS の利用を検討するとともに，本稿で触れなかったメインメモリ–SPE ローカルストア間の DMA 転送の有効利用を進める必要がある．また, SPE が持つ 128 ビット長のベクトル演算器の利用も処理の高速化に寄与するだろう．

## 参考文献

1) 情報処理学会：情報処理ハンドブック, オーム社, 1995

2) C. L. Liu: *Elements of Discrete Mathematics, 2nd Edition*, McGraw-Hill, 1985 (邦訳: C. L. リュー: 離散数学入門 (成嶋 弘, 秋山 仁 訳), マグロウヒル出版, 東京, 1986)

3) 塩田 紳二, 安田 絹子, 國司 光宣, 平 初, 川井 義治, 古坂大地, 青柳 信吾, 八重樫 剛史: PLAYSTATION 3 Linux 完全攻略ガイド, インプレスジャパン, 2007

4) 超並列計算研究会: PC クラスタ超入門〜PC クラスタ型並列計算機の基礎と講習〜, `http://www.is.doshisha.ac.jp/SMPP/report/1999/990910/`, 1999

表 4 計算時間 (SPE ライブラリ)

| ワーカ数 (個) | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 計算時間 (s) | 6.037 | 3.023 | 2.019 | 1.518 | 1.217 | 1.017 |
| 速度向上比 ∗ | 1 | 1.997 | 2.990 | 3.978 | 4.959 | 5.934 |

∗ ワーカ 1 個での計算時間/計算時間

```
def
cpi(ID, Loop)
{
  N = length(ID);
  Loop1 = idiv(Loop, N);
  Start = 0;
  Width = 1.0 / Loop;
  for (I = ID; length(I) > 1; I = cdr(I)) {
    ox_rpc(car(I), "cpi_sub", Start, Start + Loop1, Width);
    Start += Loop1;
  }
  ox_rpc(car(I), "cpi_sub", Start, Loop, Width);

  Pi = 0;
  for (I = ID; I != []; I=cdr(I)) {
    Pi += ox_pop_local(car(I));
  }
  return Pi / Loop;
}

def
cpi_sub(Start, End, Width)
{
  Pi = 0;
  for (I = Start; I < End; I++) {
    X = (I + 0.5) * Width;
    Pi += 4.0 / (1.0 + X ^ 2);
  }
  return Pi;
}

def
time_cpi(ID, Loop)
{
  T0 = time();
  print(cpi(ID, Loop));
  T1 = time();
  return T1[3]-T0[3];
}

def
rpc_cpi2_init(L)
{
  rpc_load(L, "cpi2.asir");
}

def
rpc_load(L, FN)
{
  for (I = L; I !=[]; I = cdr(I))
    ox_rpc(car(I), "load", FN);
}

end$
```

図 2 Risa/Asir を用いた $\pi$ 計算プログラム (cpi2.asir)

```
#include <stdio.h>
#include <stdlib.h>
#include <libspe2.h>
#include <pthread.h>
#include "pi.h"
#define NUM_SPE 6

typedef struct {
  spe_context_ptr_t spe_ctx;
  pi_params_t       *pi_params;
} thread_arg_t;

void *run_spe_thread(void *thread_arg)
{
  thread_arg_t *arg = (thread_arg_t *) thread_arg;
  unsigned int entry;
  spe_stop_info_t stop_info;
  entry = SPE_DEFAULT_ENTRY;
  spe_context_run(arg->spe_ctx, &entry, 0, arg->pi_params, NULL, &stop_info);
  return NULL;
}

int main(int argc, char *argv[])
{
  static pi_params_t pi_params[NUM_SPE] __attribute__((aligned(16)));
  static pi_data_t pi_data[NUM_SPE] __attribute__((aligned(16)));
  int i, ret, start, loop = 50000000, loop1;
  double width, pi;
  spe_program_handle_t *spe_prog;
  spe_context_ptr_t spe_ctx[NUM_SPE];
  pthread_t thread[NUM_SPE];
  thread_arg_t arg[NUM_SPE];

  spe_prog = spe_image_open("pi-spe.elf");
  for (i = 0; i < NUM_SPE; i++) {
    spe_ctx[i] = spe_context_create(0, NULL);
    ret = spe_program_load(spe_ctx[i], spe_prog);
  }
  loop1 = loop / NUM_SPE + 1;
  start = 0;
  width = 1.0 / loop;
  for (i = 0; i < NUM_SPE; i++) {
    pi_data[i].start = start;
    if (start + loop1 < loop)
      pi_data[i].end = start + loop1;
    else
      pi_data[i].end = loop;
    pi_data[i].width = width;
    pi_params[i].ea_in  = (unsigned long) &pi_data[i];
    pi_params[i].ea_out = (unsigned long) &pi_data[i];
    pi_params[i].size = sizeof(pi_data_t);
    arg[i].spe_ctx = spe_ctx[i];
    arg[i].pi_params = &pi_params[i];
    ret = pthread_create(&thread[i], NULL, run_spe_thread, &arg[i]);
    start += loop1;
  }
  for (i = 0; i < NUM_SPE; i++) {
    pthread_join(thread[i], NULL);
    ret = spe_context_destroy(spe_ctx[i]);
  }
  ret = spe_image_close(spe_prog);
  pi = 0.0;
  for(i = 0; i < NUM_SPE; i++)
    pi += pi_data[i].s;
  pi /= loop;
  printf("[PPE] pi = %g\n", pi);

  return 0;
}
```

図 4 SPE ライブラリを用いた $\pi$ 計算プログラム (pi-ppe.c, PPE 用，マスタ)

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "pi.h"

int main(unsigned long long spe, unsigned long long argp)
{
  static pi_params_t pi_params __attribute__((aligned(16)));
  static pi_data_t pi_data __attribute__((aligned(16)));
  int i;
  int tag = 1;
  double x;

  spu_mfcdma64(&pi_params, mfc_ea2h(argp), mfc_ea2l(argp), sizeof(pi_params_t), tag, MFC_GET_CMD);
  spu_writech(MFC_WrTagMask, 1 << tag);
  spu_mfcstat(MFC_TAG_UPDATE_ALL);

  spu_mfcdma64(&pi_data, mfc_ea2h(pi_params.ea_in), mfc_ea2l(pi_params.ea_in), pi_params.size, tag,
               MFC_GET_CMD);
  spu_writech(MFC_WrTagMask, 1 << tag);
  spu_mfcstat(MFC_TAG_UPDATE_ALL);

  pi_data.s = 0.0;
  for(i = pi_data.start; i < pi_data.end; i++) {
    x = (i + 0.5) * pi_data.width;
    pi_data.s += 4.0 / (1.0 + x * x);
  }

  spu_mfcdma64(&pi_data, mfc_ea2h(pi_params.ea_out), mfc_ea2l(pi_params.ea_out), pi_params.size, tag,
               MFC_PUT_CMD);
  spu_writech(MFC_WrTagMask, 1 << tag);
  spu_mfcstat(MFC_TAG_UPDATE_ALL);

  return 0;
}
```

図 5 SPE ライブラリを用いた $\pi$ 計算プログラム (pi-spe.c, SPE 用，ワーカ)

```
 1 last pid: 46149;  load averages:  3.82,  2.88,  3.24
 2 up 76+19:51:43  05:14:21
 3 80 processes:  5 running, 75 sleeping
 4 CPU: 96.3% user,  0.0% nice,  3.5% system,  0.2% interrupt,  0.0% idle
 5 Mem: 251M Active, 2113M Inact, 272M Wired, 33M Cache, 214M Buf, 300M
 6 Free
 7 Swap: 8192M Total, 108K Used, 8192M Free
 8
 9   PID USERNAME    THR PRI NICE   SIZE    RES STATE  C   TIME   WCPU COMMAND
10 46105 siraisi       1 118    0 36228K  9968K RUN    1   0:13 97.56% asir
11 46103 siraisi       1 117    0 36228K  9968K CPU2   2   0:15 97.07% asir
12 46101 siraisi       1 116    0 36228K  9968K CPU3   3   1:02 91.06% asir
13 46107 siraisi       1 116    0 36228K  9968K CPU0   0   0:18 90.77% asir
```

行頭の数字は，便宜上付けた行番号である．

図 7 top コマンドの実行結果 (一部)