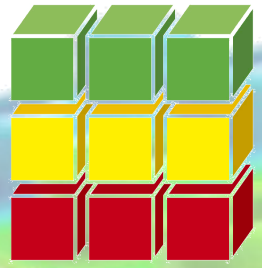


Embedded Multicore Consortium

www.embeddedmulticore.org

# マルチコアソフト開発実践編

## ～初心者がマルチコアソフト開発を成功させるポイント～

2020.11

ガイオテクノロジー(株)
浅野昌尚

Embedded Multicore Consortium

# 本題の前に(自己紹介)

## ■ 自己紹介

数十年にわたり、ず～っと組み込み開発向けのCコンパイラを開発していました。

- C言語プログラムがどのようなコードになるか、おおよそ解ります
- マイコンのアセンブリ言語は、おおよそ解っていますし、コードを読むのは好きです

マルチコアプログラミングの経験は全くありません。

- 開発したCコンパイラはマルチコア対応ではありません
- OpenMP等のマルチコア向け言語拡張については理解しています
- スレッドライブラリについての知識もあります

## ■ 初心者によるマルチコア化事例

今回の発表は、逐次プログラムをマルチコア対応に書き換える挑戦です。

- 小さなプログラムで挑戦
- 3種類の方法でマルチコア化
- OpenMPとpthreadを使用する

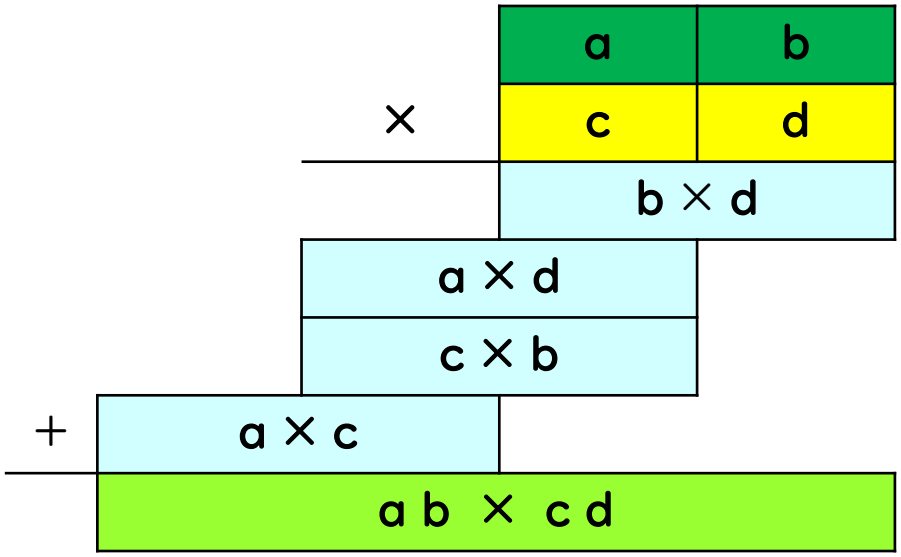
この挑戦は継続中です。
本発表内容を含め、課題や失敗例など、EMCサイトで随時公開してゆく予定です。

# 題材は演算プログラム

■ 結果が特定の値になる乗算の組み合わせを求める

- 乗算は 2バイト × 2バイト (結果は4バイト)
- 1バイト×1バイトの乗算と加算を使って計算する
- 乗算結果が0xABCDEF00になる組み合わせを10件探す

乗算回数は、約43億回
( 64k回×64k回 )




2バイトデータ
2バイトデータ
下位バイトどうしの乗算結果
上位と下位バイトの乗算結果
上位と下位バイトの乗算結果
上位バイトどうしの乗算結果
各乗算結果を加算し、4バイトの乗算結果を求める

# 逐次処理プログラム例

```
#include <stdio.h>
#define PATTERN (0x0ABCDEF00)     // 6件
#define COUNT   (10)
static int a,b,c,d,A,B,C,D,E,F,X;
static struct {int a,b;} found[COUNT];

static void mul11() { A = (a * c) << 16; }
static void mul12() { B = (a * d) << 8; }
static void mul21() { C = (c * b) << 8; }
static void mul22() { D = (b * d); }
static void addAB() { E = (A + B); }
static void addCD() { F = (C + D); }
static void addEF() { X = (E + F); }
static int  check() { return !(X ^ PATTERN); }
```

**この逐次処理プログラムを3種類の並列化処理を使って、それぞれ逐次処理の性能を超えたい**

- **データ分割による並列化**
  全体を2分割・4分割等に分けて処理する
- **ジョブ(タスク)による並列化**
  複数のジョブに細分化して並列に動かす
- **パイプライン処理による並列化**
  処理の流れを分けて、並列化する

```
void main(void)
{
   unsigned int i,j,count;

   count = COUNT;
   for ( i = 0 ; count && i <= 0xFFFF ; i++ )
      for ( j = 0 ; count && j <= 0xFFFF ; j++ )
      {
         a = i >> 8;          // 1バイト取り出す(左辺上位)
         b = i & 0xFF;        // 1バイト取り出す(左辺下位)
         c = j >> 8;          // 1バイト取り出す(右辺上位)
         d = j & 0xFF;        // 1バイト取り出す(右辺下位)
         mul22();             // 乗算1(下位×下位)
         mul12();             // 乗算2(上位×下位) << 8
         mul21();             // 乗算3(下位×上位) << 8
         mul11();             // 乗算4(上位×上位) << 16
         addCD();             // 加算1(乗算1+乗算3)
         addAB();             // 加算2(乗算2+乗算4)
         addEF();             // 加算3(加算1+加算2)
         if( check() )
         {
            count--;
            found[count].a = i;
            found[count].b = j;
            printf("%8.8X = %4.4X * %4.4X¥n",X,i,j);
         }
      }
   return;
}
```

# プログラムの動作結果 (逐次処理)

```
Cygwin@WorkPC /home
$ ls
Sample.c                          ソースプログラム

Cygwin@WorkPC /home
$ gcc Sample.c                    コンパイル

Cygwin@WorkPC /home
$ ./a.exe                         実行
ABCDEF00 = BB80 * EA92                  実行結果
ABCDEF00 = BBA8 * EA60
ABCDEF00 = C350 * E130                結果が 0xABCDEF00 となる
ABCDEF00 = E130 * C350                乗算の組み合わせを6件検出
ABCDEF00 = EA60 * BBA8
ABCDEF00 = EA92 * BB80
```

実行時間は約50秒

# 演算プログラムの並列処理化

## 逐次処理プログラムを3種類の並列化処理に書き換える

- データ分割による並列化
  全体を2分割・4分割等に分けて処理する
- ジョブ(タスク)による並列化
  複数のジョブに細分化して並列に動かす
- パイプライン処理による並列化
  処理の流れを分けて、並列化する

# データ分割による並列処理化

```
void main(void)
{
    unsigned int i,j,count;

    count = COUNT;
    for ( i = 0 ; count && i <= 0xFFFF ; i++ )
        for ( j = 0 ; count && j <= 0xFFFF ; j++ )
        {
            a = i >> 8;         // 1バイト取り出す(左辺上位)
            b = i & 0xFF;       // 1バイト取り出す(左辺下位)
            c = j >> 8;         // 1バイト取り出す(右辺上位)
            d = j & 0xFF;       // 1バイト取り出す(右辺下位)
            mul22();            // 乗算1(下位×下位)
            mul12();            // 乗算2(上位×下位) << 8
            mul21();            // 乗算3(下位×上位) << 8
            mul11();            // 乗算4(上位×上位) << 16
            addCD();            // 加算1(乗算1+乗算3)
            addAB();            // 加算2(乗算2+乗算4)
            addEF();            // 加算3(加算1+加算2)
            if( check() )
            {
                count--;
                found[count].a = i;
                found[count].b = j;
                printf("%8.8X = %4.4X * %4.4X¥n",X,i,j);
            }
        }
    return;
}
```

このfor文を区間(範囲)分け、
複数のループ作り、並列処理化

どうする？

- 内側のforループを子関数化
- 外側のforループは子関数を呼び出す
- 外側のforループをいくつかのスレッドで分割処理する

OpenMPを使ってみます

# データ分割による並列処理化

```
void sub(unsigned int i)
{
   unsigned int j;

   for ( j = 0 ; count && j <= 0xFFFF ; j++ )
   {
      a = i >> 8;          // 1バイト取り出す(左辺上位)
      b = i & 0xFF;        // 1バイト取り出す(左辺下位)
      c = j >> 8;          // 1バイト取り出す(右辺上位)
      d = j & 0xFF;        // 1バイト取り出す(右辺下位)
      mul22();             // 乗算1(下位×下位)
      mul12();             // 乗算2(上位×下位) << 8
      mul21();             // 乗算3(下位×上位) << 8
      mul11();             // 乗算4(上位×上位) << 16
      addCD();             // 加算1(乗算1+乗算3)
      addAB();             // 加算2(乗算2+乗算4)
      addEF();             // 加算3(加算1+加算2)
      if( check() )
      {
         count--;
         found[count].a = i;
         found[count].b = j;
         printf("%8.8X = %4.4X * %4.4X¥n",X,i,j);
      }
   }
   return;
}
```

```
void main(void)
{
   unsigned int i;

   count = COUNT;

#pragma omp parallel for
   for ( i = 0 ; i <= 0xFFFF ; i++ )
      sub(i);

   return;
}
```

OpenMPに任せる

- OpenMP拡張仕様無効時にはシングル動作
  gccであれば、-fopenmp オプション
- for文を適切にマルチスレッドにしてくれる
- 動いているのは8スレッド(動作環境による)
- しかし、この変更だけでは誤動作する

# データ分割による並列処理化(誤動作例)

```
Cygwin@WorkPC /home
$ ./a.exe
ABCDEF00 = BB80 * EA92
ABCDEF00 = BBA8 * EA60
ABCDEF00 = C350 * E130
ABCDEF00 = E130 * C350
ABCDEF00 = EA60 * BBA8
ABCDEF00 = EA92 * BB80
```

逐次処理の結果

検出データは6件

```
Cygwin@WorkPC /home
$ ./a.exe
ABCDEF00 = C350 * E130
ABCDEF00 = EA60 * BBA8
ABCDEF00 = EA92 * BB80
ABCDEF00 = BB80 * EA92
ABCDEF00 = BBA8 * EA60
```

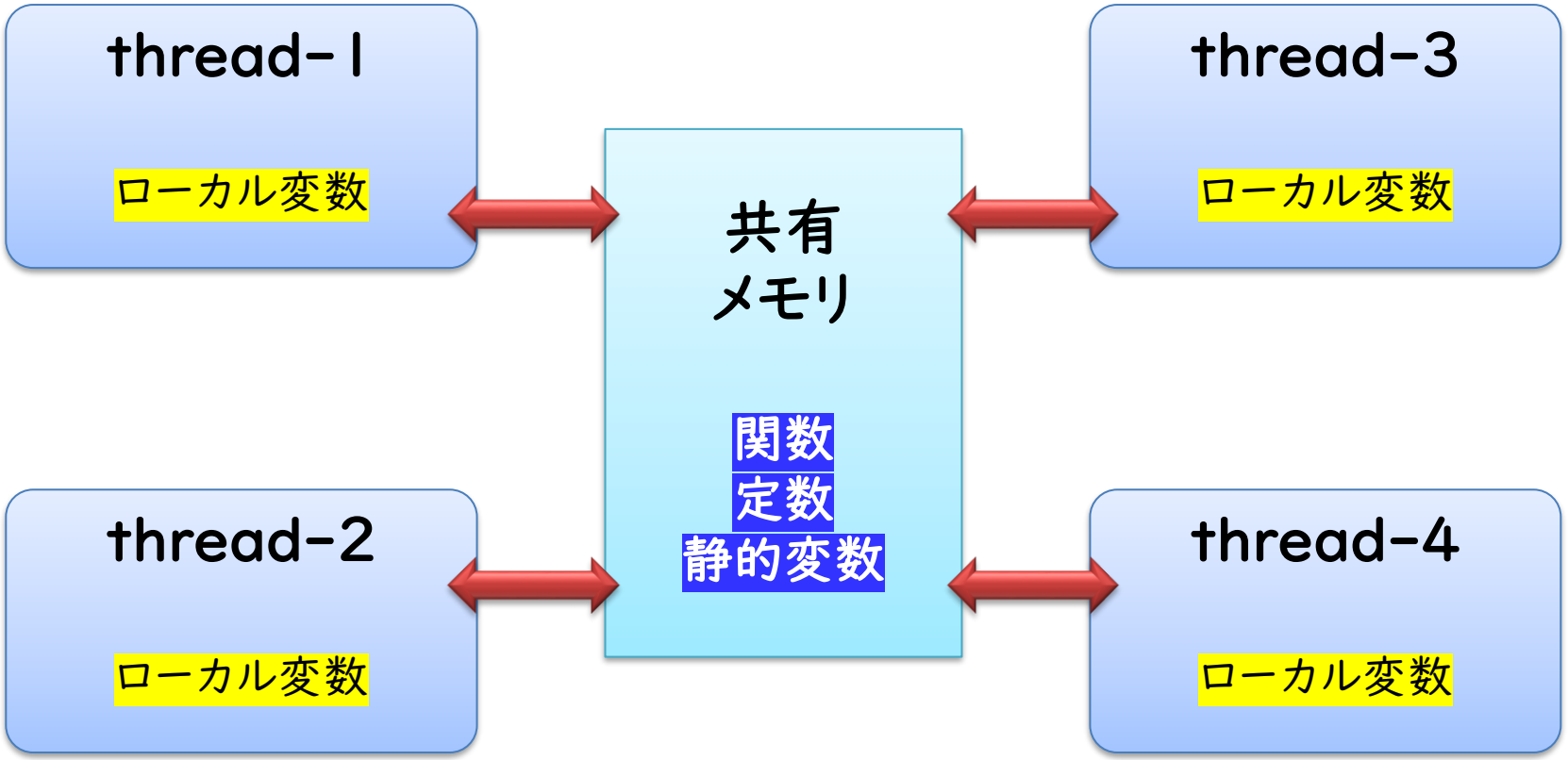
誤動作の結果

検出数が1つ足りない

誤動作は発生し難く
見逃してしまう危険性大！
(現象が発生するのは10回に1回程度)

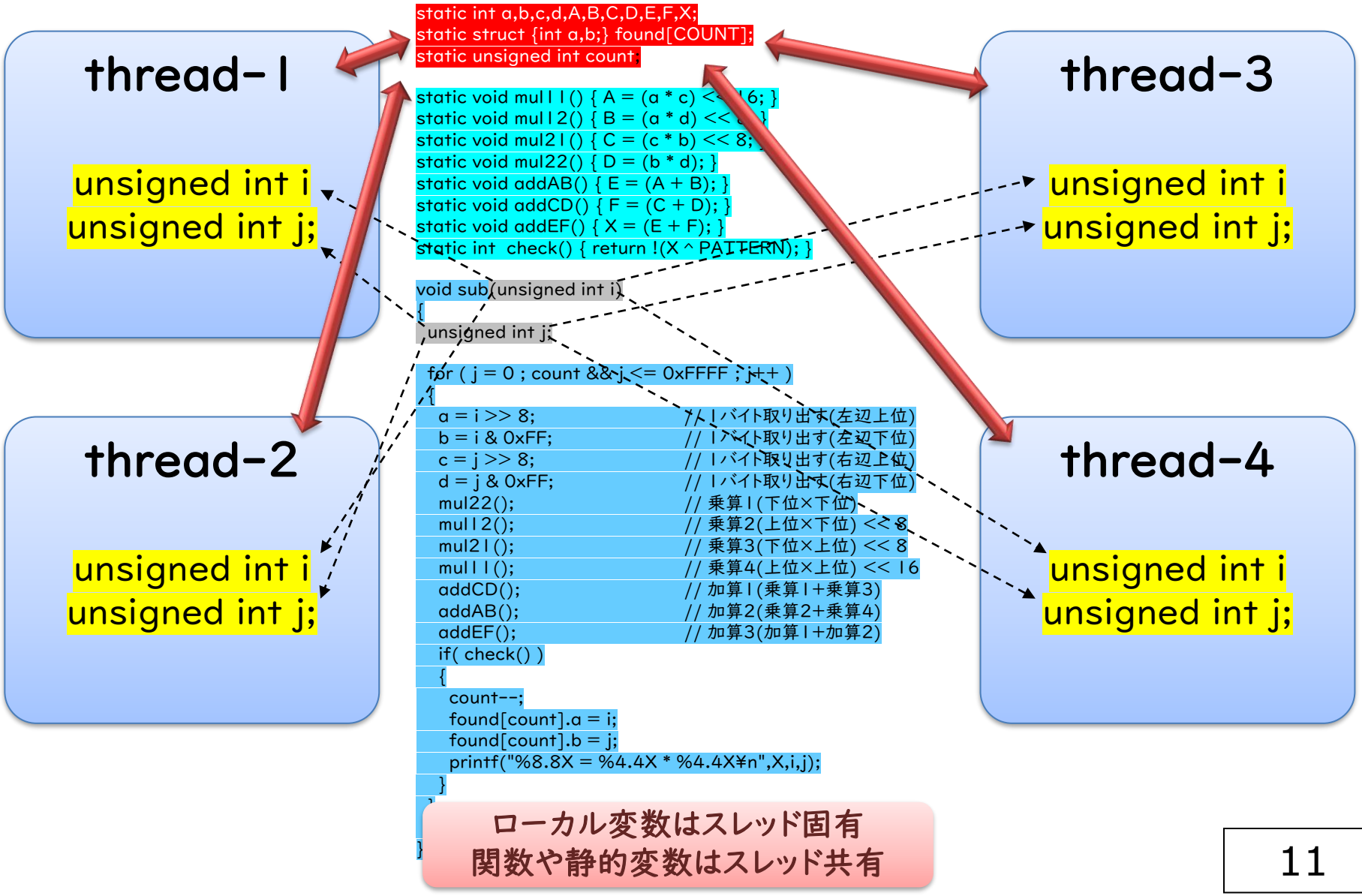
# リソースの競合(誤動作の原因)

- マルチコアで動作するプログラムでは、割り込み処理と同じようにリソース競合問題が発生する




ローカル変数はスレッド固有
関数や静的変数はスレッド共有

# データ分割による並列処理化(誤動作の原因)



# データ分割による並列処理化(競合回避)

## ■ 競合変数の回避策

- **競合変数はスタティック、グローバル変数**
  複数のスレッドがアクセスし競合する
- **可能あれば、ローカル変数化**
  関数内で閉じた利用の変数
  代入から最終参照までが関数内で終結する
  前回動作時の値を使用しない
- **不可能なものは排他制御**
  関数外でも使用される変数
  代入から最終参照までが関数内で終結しない
  前回動作時の値を継続して使用する

```
static int a,b,c,d,A,B,C,D,E,F,X;
static struct {int a,b;} found[COUNT];
static unsigned int count;

static void mul11() { A = (a * c) << 16; }
static void mul12() { B = (a * d) << 8; }
static void mul21() { C = (c * b) << 8; }
static void mul22() { D = (b * d); }
static void addAB() { E = (A + B); }
static void addCD() { F = (C + D); }
static void addEF() { X = (E + F); }
static int  check() { return !(X ^ PATTERN); }

void sub(unsigned int i)
{
  unsigned int j;

  for ( j = 0 ; count && j <= 0xFFFF ; j++ )
  {
    a = i >> 8;                 // 1バイト取り出す(左辺上位)
    b = i & 0xFF;               // 1バイト取り出す(左辺下位)
    c = j >> 8;                 // 1バイト取り出す(右辺上位)
    d = j & 0xFF;               // 1バイト取り出す(右辺下位)
    mul22();                    // 乗算1(下位×下位)
    mul12();                    // 乗算2(上位×下位) << 8
    mul21();                    // 乗算3(下位×上位) << 8
    mul11();                    // 乗算4(上位×上位) << 16
    addCD();                    // 加算1(乗算1+乗算3)
    addAB();                    // 加算2(乗算2+乗算4)
    addEF();                    // 加算3(加算1+加算2)
    if( check() )
    {
      count--;
      found[count].a = i;
      found[count].b = j;
      printf("%8.8X = %4.4X * %4.4X¥n",X,i,j);
    }
  }
  return;
}
```

# データ分割による並列処理化(競合回避)

```
void sub(unsigned int i)
{
   int a,b,c,d,A,B,C,D,E,F,X;
   unsigned int j;

   for ( j = 0 ; count && j <= 0xFFFF ; j++ )
   {
      a = i >> 8;          // 1バイト取り出す(左辺上位)
      b = i & 0xFF;        // 1バイト取り出す(左辺下位)
      c = j >> 8;          // 1バイト取り出す(右辺上位)
      d = j & 0xFF;        // 1バイト取り出す(右辺下位)
      D = mul22(b,d);      // 乗算1(下位×下位)
      B = mul12(a,d);      // 乗算2(上位×下位) << 8
      C = mul21(c,b);      // 乗算3(下位×上位) << 8
      A = mul11(a,c);      // 乗算4(上位×上位) << 16
      F = addCD(C,D);      // 加算1(乗算1+乗算3)
      E = addAB(A,B);      // 加算2(乗算2+乗算4)
      X = addEF(E,F);      // 加算3(加算1+加算2)
      if( check(X) )
      {
#pragma omp critical(crit_val)
         {
            count--;
            found[count].a = i;
            found[count].b = j;
         }
         printf("%8.8X = %4.4X * %4.4X¥n",X,i,j);
      }
   }
   return;
}
```

```
void main(void)
{
   unsigned int i;

   count = COUNT;

#pragma omp parallel for
   for ( i = 0 ; i <= 0xFFFF ; i++ )
      sub(i);

   return;
}
```

ローカル変数化

排他制御

ローカル変数化できない

# データ分割による並列処理化(動作例)

```
Cygwin@WorkPC /home
$ gcc Sample.c
Cygwin@WorkPC /home
$ ./a.exe
ABCDEF00 = BB80 * EA92
ABCDEF00 = BBA8 * EA60
ABCDEF00 = C350 * E130
ABCDEF00 = E130 * C350
ABCDEF00 = EA60 * BBA8
ABCDEF00 = EA92 * BB80
```

逐次処理の結果

検出データは6件

```
Cygwin@WorkPC /home
$ gcc -fopenmp Sample.c
Cygwin@WorkPC /home
$ ./a.exe
ABCDEF00 = E130 * C350
ABCDEF00 = C350 * E130
ABCDEF00 = EA60 * BBA8
ABCDEF00 = EA92 * BB80
ABCDEF00 = BB80 * EA92
ABCDEF00 = BBA8 * EA60
```

並列動作の結果

検出順序は異なる

実行時間は約15秒

# データ分割による並列処理化(マクロ併用)

```
void sub(unsigned int i)
{
   int a,b,c,d,A,B,C,D,E,F,X;
   unsigned int j;

   for ( j = 0 ; count && j <= 0xFFFF ; j++ )
   {
      a = i >> 8;          // 1バイト取り出す(左辺上位)
      b = i & 0xFF;        // 1バイト取り出す(左辺下位)
      c = j >> 8;          // 1バイト取り出す(右辺上位)
      d = j & 0xFF;        // 1バイト取り出す(右辺下位)
      mul22();             // 乗算1(下位×下位)
      mul12();             // 乗算2(上位×下位) << 8
      mul21();             // 乗算3(下位×上位) << 8
      mul11();             // 乗算4(上位×上位) << 16
      addCD();             // 加算1(乗算1+乗算3)
      addAB();             // 加算2(乗算2+乗算4)
      addEF();             // 加算3(加算1+加算2)
      if( check() )
      {
#pragma omp critical(crit_val)
         {
            count--;
            found[count].a = i;
            found[count].b = j;
         }
         printf("%8.8X = %4.4X * %4.4X¥n",X,i,j);
      }
   }
   return;
}
```

```
#define mul11() ( A = (a * c) << 16 )
#define mul12() ( B = (a * d) << 8 )
#define mul21() ( C = (c * b) << 8 )
#define mul22() ( D = (b * d) )
#define addAB() ( E = (A + B) )
#define addCD() ( F = (C + D) )
#define addEF() ( X = (E + F) )
#define check() ( !(X ^ PATTERN) )
```

OpenMPに任せる(さらに高速化)

- 末端関数をマクロにしてみたら、爆速
- コードサイズには注意

# 演算プログラムの並列処理化

**逐次処理プログラムを3種類の並列化処理に書き換える**

- データ分割による並列化
  全体を2分割・4分割等に分けて処理する
- **ジョブ(タスク)による並列化**
  **複数のジョブに細分化して並列に動かす**
- パイプライン処理による並列化
  処理の流れを分けて、並列化する

# ジョブ(タスク)による並列処理化

```
void main(void)
{
    unsigned int i,j,count;

    count = COUNT;
    for ( i = 0 ; count && i <= 0xFFFF ; i++ )
        for ( j = 0 ; count && j <= 0xFFFF ; j++ )
        {
            a = i >> 8;         // 1バイト取り出す(左辺上位)
            b = i & 0xFF;       // 1バイト取り出す(左辺下位)
            c = j >> 8;         // 1バイト取り出す(右辺上位)
            d = j & 0xFF;       // 1バイト取り出す(右辺下位)
            mul22();            // 乗算1(下位×下位)
            mul12();            // 乗算2(上位×下位) << 8
            mul21();            // 乗算3(下位×上位) << 8
            mul11();            // 乗算4(上位×上位) << 16
            addCD();            // 加算1(乗算1+乗算3)
            addAB();            // 加算2(乗算2+乗算4)
            addEF();            // 加算3(加算1+加算2)
            if( check() )
            {
                count--;
                found[count].a = i;
                found[count].b = j;
                printf("%8.8X = %4.4X * %4.4X¥n",X,i,j);
            }
        }
    return;
}
```

内側のループ処理をジョブ化、
複数のジョブを並列処理

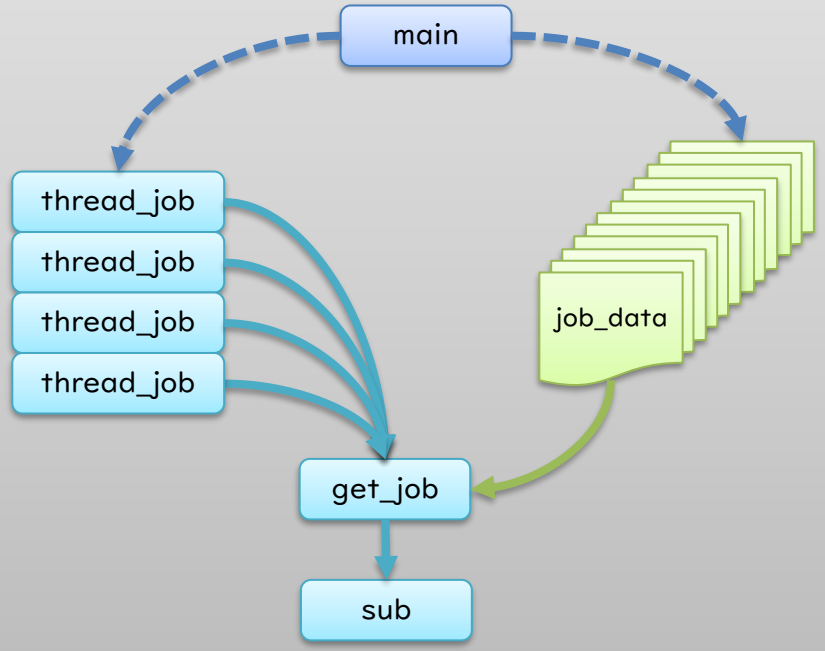
どうする？

- 内側のforループを子関数化し、ワーカースレッドとして複数動かす
- 外側のforループはジョブデータ化する
- 複数のワーカースレッドは連携してジョブデータを順次処理する

# ジョブ(タスク)による並列処理化

## pthreadによる実装構造

- mainがジョブに与えるジョブデータを作成し、THREAD数(4)のジョブ関数を起動
- ジョブ関数はジョブデータが無くなるまで、ジョブサイズ単位の処理を繰り返す




```
void main(void)
{
    unsigned int i,j;
    pthread_t    jobs[THREAD];

    for ( i = j = 0 ; j <= 0xFFFF ; j+=JOB_SIZE )
        job_data[i++] = j;
    job_counts = i;
    job_index = 0;
    pthread_mutex_init(&mutex_job, NULL);
    pthread_mutex_init(&mutex_count, NULL);

    for ( i = 0 ; i < THREAD ; i++ )
    {
        pthread_create(&jobs[i], NULL, (void *)thread_job,
                                    (void *)&job_size);
    }

    for ( i = 0; i < THREAD ; i++ )
        pthread_join(jobs[i], NULL);
    return;
}
```

# ジョブ(タスク)による並列処理化

```
void thread_job(void *arg)
{
    unsigned int *jobp;

    while ( (jobp = get_job()) != NULL )
        sub(*jobp,*((unsigned int *)arg));
}

unsigned int *get_job()
{
    int index;

    pthread_mutex_lock(&mutex_job);
    if (job_index + 1 == job_counts)
    {
        pthread_mutex_unlock(&mutex_job);
        return NULL;
    }
    index = job_index++;
    pthread_mutex_unlock(&mutex_job);
    return &job_data[index];
}
```

終わらせ方が難しい

- 10件のデータを検出した時、即座に複数のジョブを終わらせることができるか

```
void sub(unsigned int i,unsigned int job_count)
{
    int a,b,c,d,A,B,C,D,E,F,X;
    unsigned int j;
    do
    {
        for ( j = 0 ; count && j <= 0xFFFF ; j++ )
        {
            a = i >> 8;         // 1バイト取り出す(左辺上位)
            b = i & 0xFF;       // 1バイト取り出す(左辺下位)
            c = j >> 8;         // 1バイト取り出す(右辺上位)
            d = j & 0xFF;       // 1バイト取り出す(右辺下位)
            D = mul22(b,d); // 乗算1(下位×下位)
            B = mul12(a,d); // 乗算2(上位×下位) << 8
            C = mul21(c,b); // 乗算3(下位×上位) << 8
            A = mul11(a,c); // 乗算4(上位×上位) << 16
            F = addCD(C,D);// 加算1(乗算1+乗算3)
            E = addAB(A,B); // 加算2(乗算2+乗算4)
            X = addEF(E,F); // 加算3(加算1+加算2)
            if( check(X) )
            {
                pthread_mutex_lock(&mutex_count);
                count--;
                found[count].a = i;
                found[count].b = j;
                pthread_mutex_unlock(&mutex_count);
                printf("%8.8X = %4.4X * %4.4X¥n",X,i,j);
            }
        }
        i++;
    } while ( --job_count );
    return;
}
```

# ジョブ(タスク)による並列処理化(動作例)

```
Cygwin@WorkPC /home
$ gcc Sample.c
Cygwin@WorkPC /home
$ ./a.exe
ABCDEF00 = BB80 * EA92
ABCDEF00 = BBA8 * EA60
ABCDEF00 = C350 * E130
ABCDEF00 = E130 * C350
ABCDEF00 = EA60 * BBA8
ABCDEF00 = EA92 * BB80
```

逐次処理の結果

検出データは6件

```
Cygwin@WorkPC /home
$ gcc -lpthread Sample.c
Cygwin@WorkPC /home
$ ./a.exe
ABCDEF00 = BB80 * EA92
ABCDEF00 = BBA8 * EA60
ABCDEF00 = C350 * E130
ABCDEF00 = E130 * C350
ABCDEF00 = EA60 * BBA8
ABCDEF00 = EA92 * BB80
```

並列動作の結果

検出順序は同じ

実行時間は約15秒

# 演算プログラムの並列処理化

**逐次処理プログラムを3種類の並列化処理に書き換える**

- データ分割による並列化
  全体を2分割・4分割等に分けて処理する
- ジョブ(タスク)による並列化
  複数のジョブに細分化して並列に動かす
- **パイプライン処理による並列化**
  **処理の流れを分けて、並列化する**

# パイプライン処理による並列処理化

```
void main(void)
{
    unsigned int i,j,count;

    count = COUNT;
    for ( i = 0 ; count && i <= 0xFFFF ; i++ )
        for ( j = 0 ; count && j <= 0xFFFF ; j++ )
        {
            a = i >> 8;          // 1バイト取り出す(左辺上位)
            b = i & 0xFF;        // 1バイト取り出す(左辺下位)
            c = j >> 8;          // 1バイト取り出す(右辺上位)
            d = j & 0xFF;        // 1バイト取り出す(右辺下位)
            mul22();             // 乗算1(下位×下位)
            mul12();             // 乗算2(上位×下位) << 8
            mul21();             // 乗算3(下位×上位) << 8
            mul11();             // 乗算4(上位×上位) << 16
            addCD();             // 加算1(乗算1+乗算3)
            addAB();             // 加算2(乗算2+乗算4)
            addEF();             // 加算3(加算1+加算2)
            if( check() )
            {
                count--;
                found[count].a = i;
                found[count].b = j;
                printf("%8.8X = %4.4X * %4.4X¥n",X,i,j);
            }
        }
    return;
}
```

同色内の順序性無し、
色分け処理をパイプライン処理化

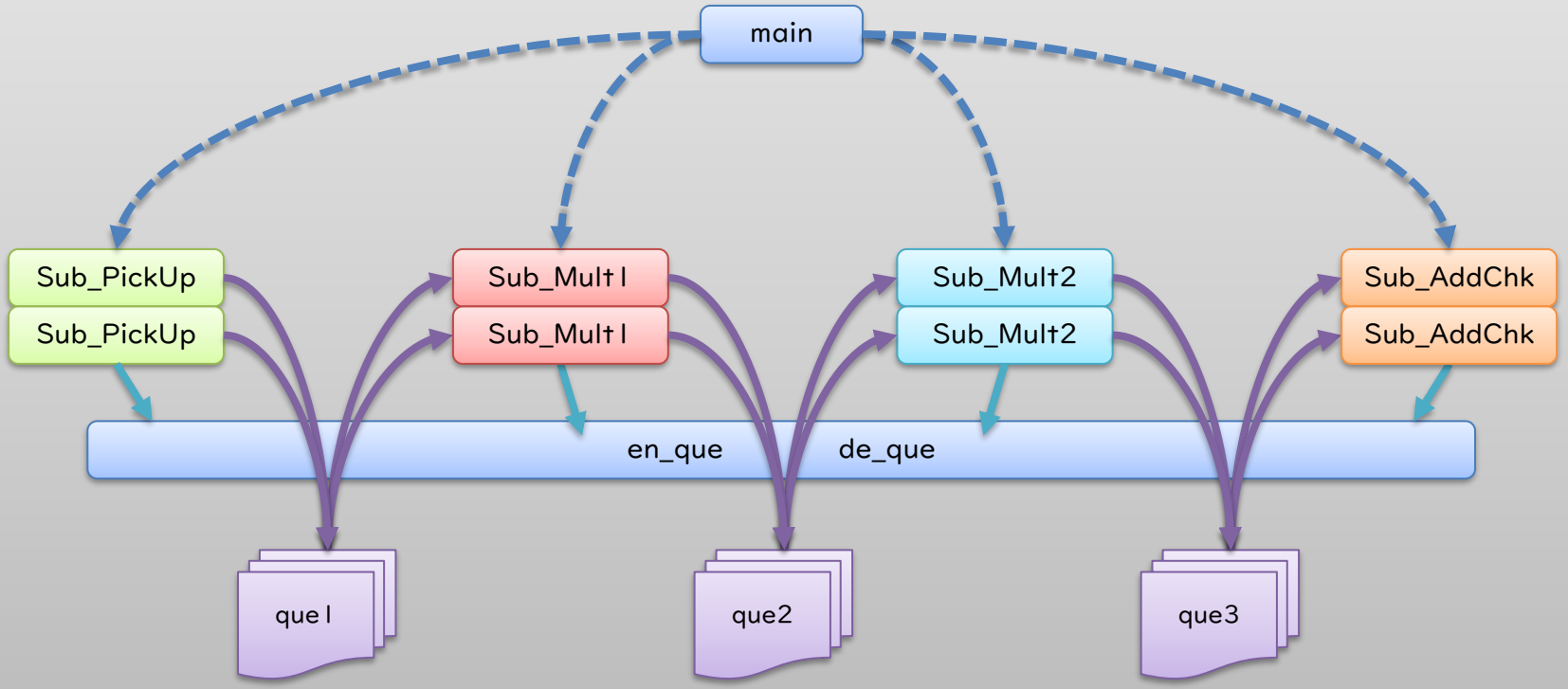
どうする？

- 色分け処理を子関数化(4×n のスレッド)
- 各色のスレッド処理は先行する色のスレッド処理が生成したデータを使って処理をする
- 連行スレッドと後続スレッドはキューで繋ぐ

# パイプライン処理による並列処理化

## pthreadによる実装構造

- 処理を4分割し、各分割処理を行うスレッドを2つずつ動かす
- 4分割の各処理はキューで繋がり、前処理データを使って処理した結果を後処理に渡す



# パイプライン処理による並列処理化

## 並列化の仕組み

- RISCプロセッサの命令動作のようにパイプライン処理で並列化する
- 各パイプラインステージでは4×2の処理関数が動き、理論上8件のデータを同時処理する

| 処理件数 | パイプラインステージ |
|---|---|
| 1 | Sub_PickUp → Sub_Mult1 → Sub_Mult2 → Sub_AddChk |
| 2 | Sub_PickUp → Sub_Mult1 → Sub_Mult2 → Sub_AddChk |
| 3 | Sub_PickUp → Sub_Mult1 → Sub_Mult2 → Sub_AddChk |
| 4 | Sub_PickUp → Sub_Mult1 → Sub_Mult2 → Sub_AddChk |
| 5 | Sub_PickUp → Sub_Mult1 → Sub_Mult2 → Sub_AddChk |
| 6 | Sub_PickUp → Sub_Mult1 → Sub_Mult2 → Sub_AddChk |
| 7 | Sub_PickUp → Sub_Mult1 → Sub_Mult2 → Sub_AddChk |
| 8 | Sub_PickUp → Sub_Mult1 → Sub_Mult2 → Sub_AddChk |
| 9 | Sub_PickUp → Sub_Mult1 → Sub_Mult2 |
| 10 | Sub_PickUp → Sub_Mult1 → Sub_Mult2 |
| : | |

# パイプライン処理による並列処理化(参考)

```
void main(void)
{
    int       i;
    pthread_t  PickUp[THREAD];
    pthread_t  Mult1[THREAD];
    pthread_t  Mult2[THREAD];
    pthread_t  AddChk[THREAD];
    type_arg   PUarg[THREAD];
    type_arg   M1arg[THREAD];
    type_arg   M2arg[THREAD];
    type_arg   ACarg[THREAD];
    type_queue que1,que2,que3;
    // キューテーブルの初期化
    que1 = que2 = que3 = que_init;
    pthread_mutex_init(&que1.mutex, NULL);
    pthread_cond_init(&que1.not_full, NULL);
    pthread_cond_init(&que1.not_empty, NULL);
    pthread_mutex_init(&que2.mutex, NULL);
    pthread_cond_init(&que2.not_full, NULL);
    pthread_cond_init(&que2.not_empty, NULL);
    pthread_mutex_init(&que3.mutex, NULL);
    pthread_cond_init(&que3.not_full, NULL);
    pthread_cond_init(&que3.not_empty, NULL);
    pthread_mutex_init(&mutex_loop, NULL);
    // PickUp_thread 生成
    for (i = 0; i < THREAD; i++)
    {
        PUarg[i].id = i;
        PUarg[i].que_in  = 0;
        PUarg[i].que_out = &que1;
        pthread_create(&PickUp[i], NULL,
        (void*)Sub_PickUp,(void*)&PUarg[i]);
    }
    // Mult1_thread 生成
    for (i = 0; i < THREAD; i++)
    {
        M1arg[i].id = i;
        M1arg[i].que_in  = &que1;
        M1arg[i].que_out = &que2;
        pthread_create(&Mult1[i], NULL,
        (void*)Sub_Mult1,(void*)&M1arg[i]);
    }
    // Mult2_thread 生成
    for (i = 0; i < THREAD; i++)
    {
        M2arg[i].id = i;
        M2arg[i].que_in  = &que2;
        M2arg[i].que_out = &que3;
        pthread_create(&Mult2[i], NULL,
        (void*)Sub_Mult2,(void*)&M2arg[i]);
    }
    // AddChk_thread 生成
    for (i = 0; i < THREAD; i++)
    {
        ACarg[i].id = i;
        ACarg[i].que_in  = &que3;
        ACarg[i].que_out = 0;
        pthread_create(&AddChk[i], NULL,
        (void*)Sub_AddChk,(void*)&ACarg[i]);
    }
    // AddChk_thread の終了待ち
    for (i = 0; i < THREAD; i++)
        pthread_join(AddChk[i], NULL);
    return;
}
```

# パイプライン処理による並列処理化(参考)

```
void Sub_PickUp(void *p)
{
   type_arg *arg = (type_arg *)p;
   type_data data;
   int i,j;

   data.stop = 0;
   for ( ; ; )
   {
      if ( !count )
      return;
      pthread_mutex_lock(&mutex_loop);
      j = Loop2++;
      i = Loop1;
      if ( i > 0xFFFF )
         break;
      if ( j == 0xFFFF )
         Loop1++,Loop2=0;
      pthread_mutex_unlock(&mutex_loop);
      data.i = i;
      data.j = j;
      data.a = i >> 8;        // 1バイト取り出す(左辺上位)
      data.b = i & 0xFF;      // 1バイト取り出す(左辺下位)
      data.c = j >> 8;        // 1バイト取り出す(右辺上位)
      data.d = j & 0xFF;      // 1バイト取り出す(右辺下位)
      en_que(arg->que_out,&data);
   }
   data.stop = STOP;
   en_que(arg->que_out,&data);
   return;
}
```

```
void Sub_Mult1(void *p)
{
   type_arg *arg = (type_arg *)p;
   type_data data;

   for ( ; count ; )
   {
      de_que(arg->que_in, &data);
      data.D = mul22(data.b,data.d); // 乗算1(下位×下位)
      data.B = mul12(data.a,data.d); // 乗算2(上位×下位) << 8
      en_que(arg->que_out,&data);
      if ( data.stop == STOP )
         break;
   }
   return;
}
void Sub_Mult2(void *p)
{
   type_arg *arg = (type_arg *)p;
   type_data data;

   for ( ; count ; )
   {
      de_que(arg->que_in, &data);
      data.C = mul21(data.c,data.b); // 乗算3(下位×上位) << 8
      data.A = mul11(data.a,data.c); // 乗算4(上位×上位) << 16
      en_que(arg->que_out,&data);
      if ( data.stop == STOP )
         break;
   }
   return;
}
```

# パイプライン処理による並列処理化(参考)

```
void Sub_AddChk(void *p)
{
    type_arg *arg = (type_arg *)p;
    pthread_mutex_t mutex;
    type_data data;
    unsigned int X;

    for ( ; count ; )
    {
        de_que(arg->que_in, &data);
        data.F = addCD(data.C,data.D);// 加算1(乗算1+乗算3)
        data.E = addAB(data.A,data.B);// 加算2(乗算2+乗算4)
        X      = addEF(data.E,data.F);     // 加算3(加算1+加算2)
        if( check(X) )
        {
            count--;
            pthread_mutex_lock(&mutex);
            found[count].a = data.i;
            found[count].b = data.j;
            pthread_mutex_unlock(&mutex);
            printf("%8.8X = %4.4X * %4.4X¥n",X,data.i,data.j);
        }
        if ( data.stop == STOP )
            break;
    }
    return;
}
```

```
// キューの空きを待って、データをキューに追加する
void en_que(type_queue *q, type_data *datap)
{
    pthread_mutex_lock(&q->mutex);
    while (q->counts == MAX_QUEUE_NUM)
        pthread_cond_wait(&q->not_full, &q->mutex);
    q->data[q->indx_w] = *datap;
    q->indx_w++;
    if (q->indx_w == MAX_QUEUE_NUM)
        q->indx_w = 0;
    q->counts++;
    pthread_cond_signal(&q->not_empty);
    pthread_mutex_unlock(&q->mutex);
}

// キューに値があればデータを取り出し、無ければ待つ
void de_que(type_queue *q, type_data *datap)
{
    pthread_mutex_lock(&q->mutex);
    while (q->counts == 0)
        pthread_cond_wait(&q->not_empty, &q->mutex);
    *datap = q->data[q->indx_r];
    q->indx_r++;
    if (q->indx_r == MAX_QUEUE_NUM)
        q->indx_r = 0;
    q->counts--;
    pthread_cond_signal(&q->not_full);
    pthread_mutex_unlock(&q->mutex);
}
```

# パイプライン処理による並列処理化(動作例)

```
Cygwin@WorkPC /home
$ gcc Sample.c
Cygwin@WorkPC /home
$ ./a.exe
ABCDEF00 = BB80 * EA92
ABCDEF00 = BBA8 * EA60
ABCDEF00 = C350 * E130
ABCDEF00 = E130 * C350
ABCDEF00 = EA60 * BBA8
ABCDEF00 = EA92 * BB80
```

逐次処理の結果

検出データは6件

```
Cygwin@WorkPC /home
$ gcc -lpthread Sample.c
Cygwin@WorkPC /home
$ ./a.exe
ABCDEF00 = BB80 * EA92
ABCDEF00 = BBA8 * EA60
ABCDEF00 = C350 * E130
ABCDEF00 = E130 * C350
ABCDEF00 = EA60 * BBA8
ABCDEF00 = EA92 * BB80
```

並列動作の結果

検出順序は同じ

実行時間は約18時間半

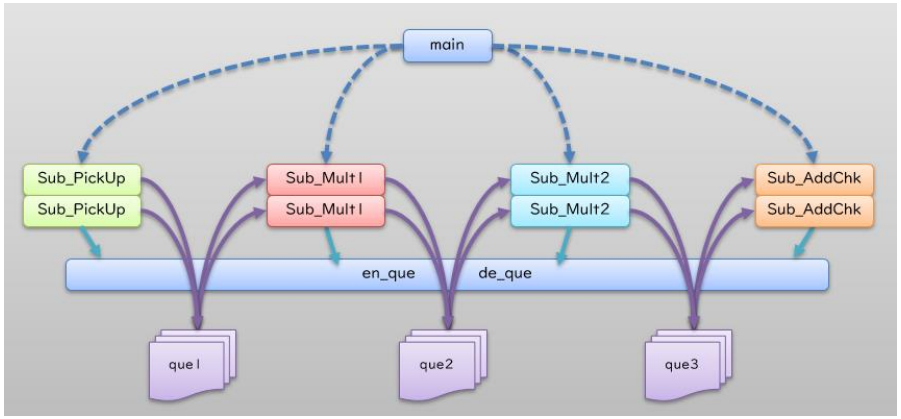
# パイプライン処理による並列処理化

**終わらせ方が難しい**

- 規定数到達時に、進行中のパイプラインを停止させる

**処理バランス(負荷分散)の調整が難しい**

- 処理負担を均等にしたい
- 4×2でなくても良い




| 処理件数 | パイプラインステージ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | Sub_PickUp | Sub_Mult1 | Sub_Mult2 | Sub_AddChk | | | | |
| 2 | Sub_PickUp | Sub_Mult1 | Sub_Mult2 | Sub_AddChk | | | | |
| 3 | | Sub_PickUp | Sub_Mult1 | Sub_Mult2 | Sub_AddChk | | | |
| 4 | | Sub_PickUp | Sub_Mult1 | Sub_Mult2 | Sub_AddChk | | | |
| 5 | | | Sub_PickUp | Sub_Mult1 | Sub_Mult2 | Sub_AddChk | | |
| 6 | | | Sub_PickUp | Sub_Mult1 | Sub_Mult2 | Sub_AddChk | | |
| 7 | | | | Sub_PickUp | Sub_Mult1 | Sub_Mult2 | Sub_AddChk | |
| 8 | | | | Sub_PickUp | Sub_Mult1 | Sub_Mult2 | Sub_AddChk | |
| 9 | | | | | Sub_PickUp | Sub_Mult1 | Sub_Mult2 | |
| 10 | | | | | Sub_PickUp | Sub_Mult1 | Sub_Mult2 | |

# パイプライン処理バランスの確認
## プロファイルを採ってみた

Flat profile:

Each sample counts as 0.01 seconds.

| % time | cumulative seconds | self seconds | calls | self ns/call | total ns/call | name |
|---|---|---|---|---|---|---|
| 46.22 | 1.04 | 1.04 | | | | _mcount_private |
| 14.67 | 1.37 | 0.33 | 4666380 | 70.72 | 70.72 | en_que |
| 14.22 | 1.69 | 0.32 | 4681968 | 68.35 | 68.35 | de_que |
| 8.00 | 1.87 | 0.18 | | | | __fentry__ |
| 6.67 | 2.02 | 0.15 | | | | Sub_PickUp |
| 4.89 | 2.13 | 0.11 | | | | Sub_Mult1 |
| 3.11 | 2.20 | 0.07 | | | | Sub_AddChk |
| 1.78 | 2.24 | 0.04 | | | | Sub_Mult2 |
| 0.44 | 2.25 | 0.01 | | | | mul12 |
| 0.00 | 2.25 | 0.00 | | | | __gcc_deregister_frame |

キューの登録数が
とても多そう

1つのキューに登録する
データ件数が問題か

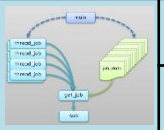
# パイプライン並列化の測定結果

■ パイプライン処理による並列化の最高速値は？

| 一度に処理するデータ数 | スレッド構成 | 処理速度 |
|---|---|---|
| 1 | 4×2 | 18:38:45.00 |
| 256 | 4×1 | 1:43.07 |
| | 4×2 | 4:36.19 |
| 1024 | 4×1 | 51.83 |
| | 4×2 | 1:00.43 |
| 4096 | 4×1 | 52.43 |
| | 4×2 | 41.08 |
| 8192 | 4×1 | 2:20.39 |
| | 4×2 | 2:17.60 |
| 65536 | 4×1 | 1:57.27 |
| | 4×2 | 1:34.11 |

粒度
と呼ぶらしい

# 測定結果(まとめ)

| 並列分類 | スレッド構成等 | 補足情報<br>(job/queueのデータ数) | 処理速度 |
|---|---|---|---|
| 逐次処理 | | | 50.07 |
| | マクロ化 | | 21.38 |
| OpenMP | 8 | | 15.27 |
| | 8+マクロ化 | | 5.97 |
| ジョブ(タスク)<br>並列化  | 8 | 256 | 15.95 |
| | 8+マクロ化 | 256 | 5.83 |
| パイプライン<br>並列化  | 4×1 | 4096 | 52.43 |
| | 4×2 | 4096 | 41.08 |

- OpenMPは並列化が容易 (ソース変更が少ない)
- ジョブ(タスク)並列化でも同等の性能が出せる
- パイプライン並列化は未知
  - 更に改善する可能性はありそうだが、そのための情報を揃えるのが難しそう
    プロファイラのデータは少し怪しい(マルチスレッドについては不正確か)
- ツールは不十分
  - 並列化のための計測ツール
  - 競合変数に関する情報提供ツール
  - スレッドライブラリの補助ツール(使用ミスの警告等)

# 測定結果(参考)

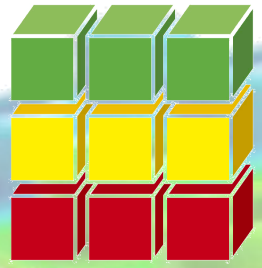
■ 計測環境

Cygwin64 ( Windows10 Pro )

| | | |
|---|---|---|
| Intel Core i7-7700 | | 3.60GHz |
| | Core | 4 |
| | Thread | 8 |
| | Mem | 16GB |

# 挑戦は継続中

- **挑戦状況は随時公開予定**
  https://www.embeddedmulticore.org/ にて



Embedded
Multicore
Consortium

www.embeddedmulticore.org

# お問い合わせは

www.embeddedmulticore.org

Embedded
Multicore
Consortium