プログラミング環境特論

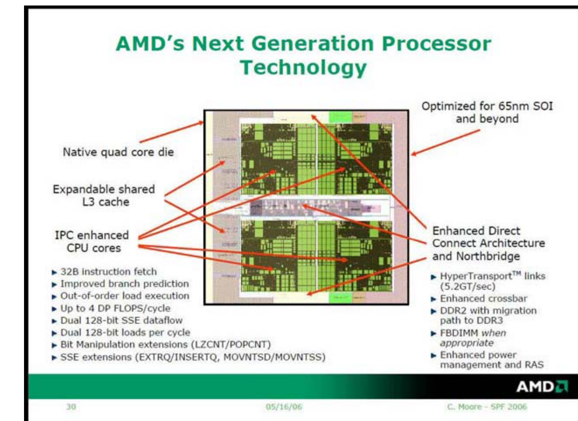
組み込みシステムでの
マルチコアプロセッサのプログラミングと課題

佐藤　三久

筑波大学

プロセッサ研究開発の動向

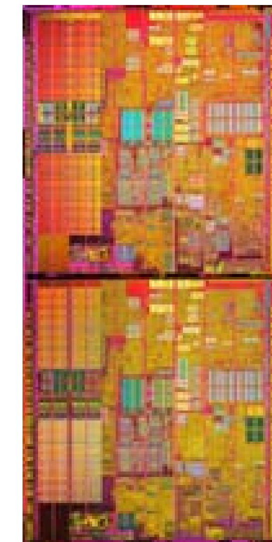
- クロックの高速化、製造プロセスの微細化
  - いまでは3GHz, 数年のうちに10GHzか！？
    - インテルの戦略の転換 ⇒ マルチコア
    - クロックは早くならない！？
  - プロセスは65nm⇒45nm , 将来的には32nm
    - トランジスタ数は増える！



Good news & bad news!

- アーキテクチャの改良
  - スーパーパイプライン、スーパースカラ、VLIW…
  - キャッシュの多段化、マイクロプロセッサでもL3キャッシュ
  - マルチスレッド化、Intel Hyperthreading
    - 複数のプログラムを同時に処理
  - マルチコア：1つのチップに複数のCPU

プログラミングに関係するところ



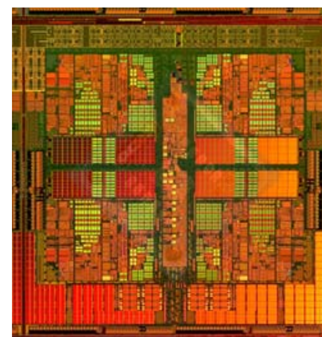
インテル® Pentium® プロセッサ エクストリーム・エディションのダイ

マルチコア・プロセッサの分類

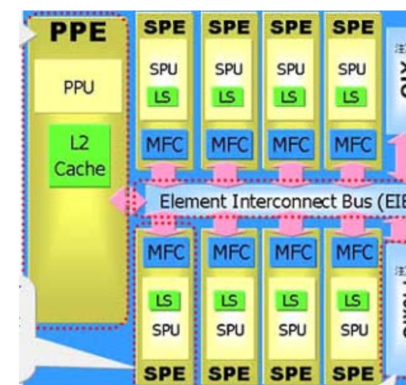
	SMP（ Symmetric Multi Processor) 同じ種類のコア	AMP（ Asymmetric Multi Processor) 異種のコアが混在
共有メモリ	サーバー用マルチコア MPCore ルネサス　M32R ルネサス　RP1 (SH3X)	（コミュニケーションのための少量のメモリがついていることがある）
分散メモリ	富士通　FR-V(?)	IBM Cell DSP 混載チップ 最近では、GPU混載も

SMPとAMP

- SMP（Symmetric Multi Processor）
  - 同じコアを複数配置したもの
  - 普通は、共有メモリ型
  - 汎用

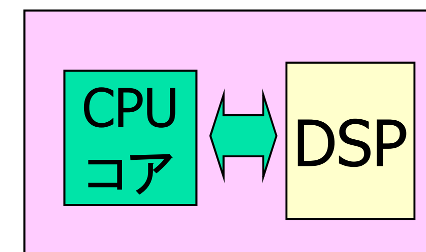


AMD quad-core



IBM Cell

- AMP（Asymmetric Multi Processor）
  - 機能が異なるコアを配置した非対称な型
  - 通常は、分散メモリ型
  - Cell プロセッサが有名
  - GPUやDSPもこれに分類
  - 機能が特化されている　⇒　コストが安い

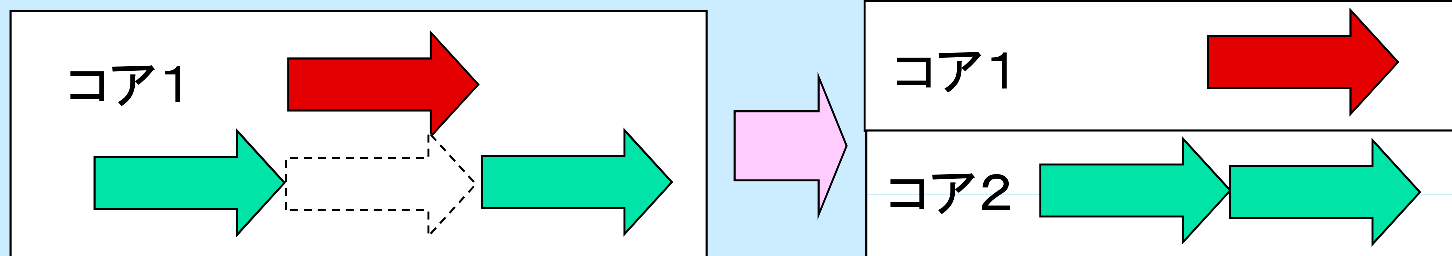


- 共有メモリ　vs. 分散メモリ
  - 各コアからどのようにメイン・メモリにアクセスできるかもプログラミングを考える場合に重要な点

マルチコアプロセッサの使い方 （その1）

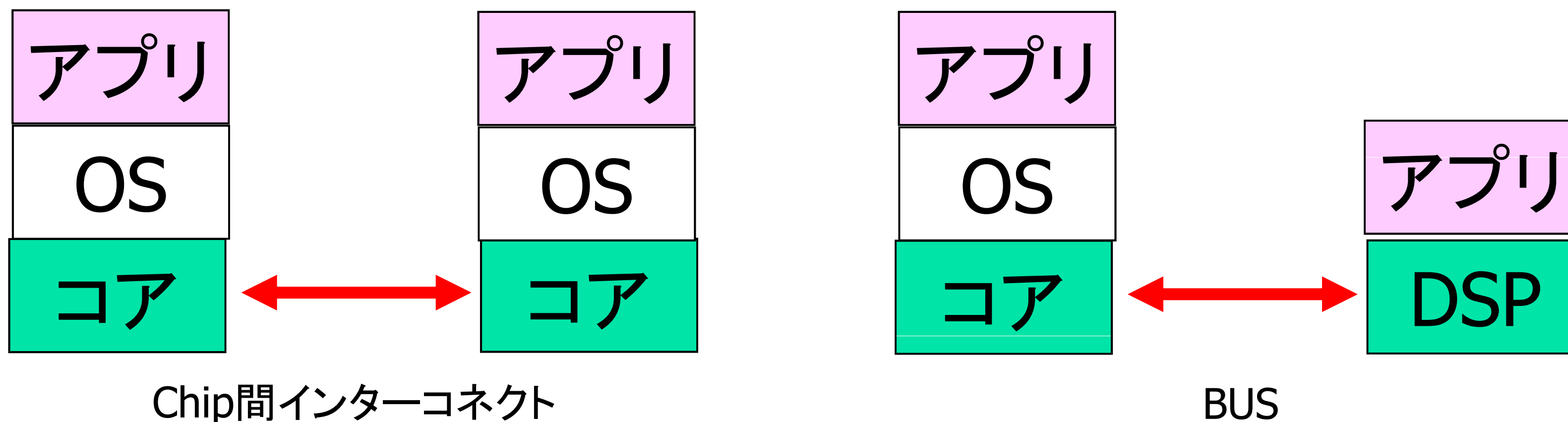
- 複数のプロセス・スレッドでの利用
  - 主に、共有メモリSMPのマルチコアプロセッサ
  - 通常、組み込みシステムはマルチタスク（プロセス）のプログラム
  - 特別なプログラミングはいらない（はず）

- シングルコアでのマルチタスクプログラムが、マルチコア（SMP)で動かない？
  - シングルコアで優先度が高いタスクの実行中は優先度が低いタスクが動かないことが前提としている場合
  - マルチコアでは＊本当に＊並列に動いてしまい、優先度が無効になる
  - キチンと同期をとりましょう。



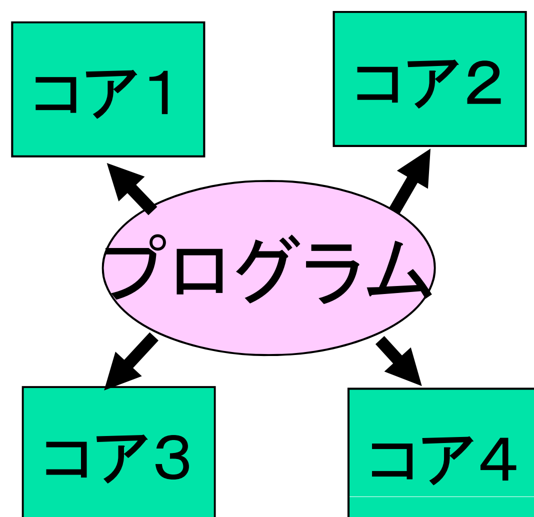
マルチコアプロセッサの使い方(その2)

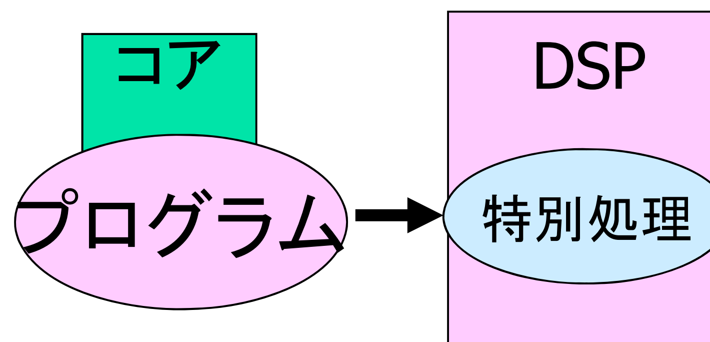
- コアごとに異なる機能で使う
  - AMP型では主な使い方
    - SMP型でも、共有メモリを持たない場合にはこのタイプ
    - 従来、複数のチップで構成したものを1つに
  - 個々のコアにOSを走らせる。(DSPなどOSがない場合もあり)
    - 違うOSの場合も。LinuxとRTOS
    - SMPで、VM等を使って違うOSを乗せる場合も同じ
  - 通信はChip内インタコネクトまたはバス
    - RPCモデルも使える



マルチコアプロセッサの使い方(3)

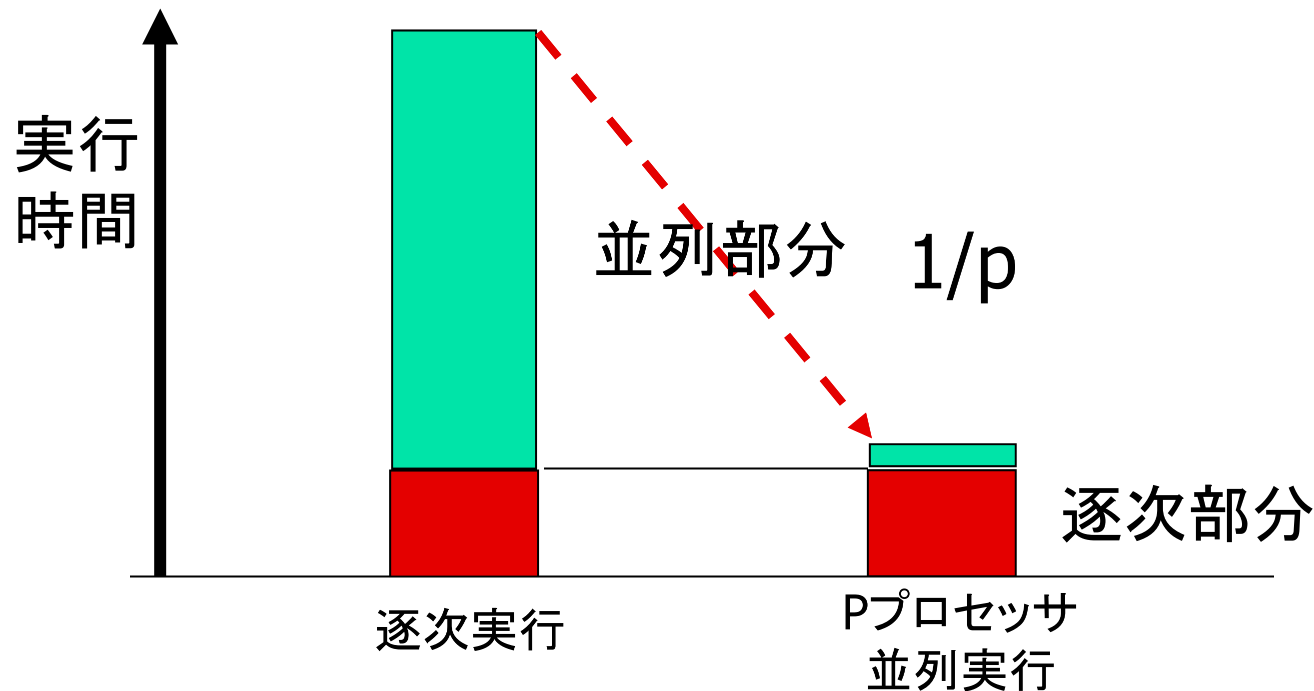
- 高性能化のためにつかう（最終的にはみんなこれ？！）
  - 複数のコアで並列処理
  - 共有メモリSMPの場合はOpenMP
    - ハイエンドで使われている技術が使える
  - AMPでも、DSP等を加速機構としてつかっている場合は、このケースに当たる。





並列処理による高速化:「アムダールの法則」

- アムダールの法則
  - 逐次処理での実行時間をT_1, 逐次で実行しなくてはならない部分の比率がαである場合、pプロセッサを用いて実行した時の実行時間(の下限)T_pは、$T_p = \alpha * T_1 + (1-\alpha) * T_1/p$
  - 逐次部分があるため、高速化には限界があるということ。



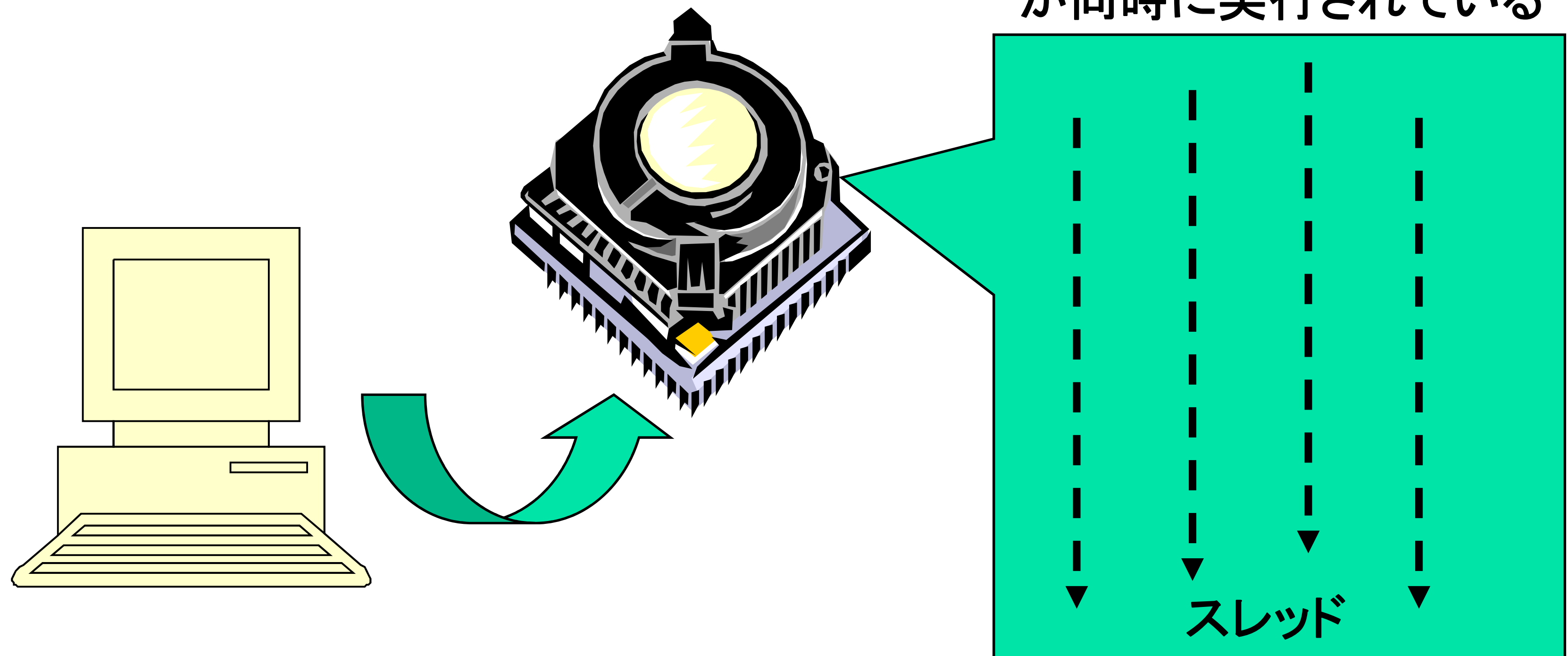
並列プログラミング・モデル

- メッセージ通信 (Message Passing)
  - メッセージのやり取りでやり取りをして、プログラムする
  - 分散メモリシステム（共有メモリでも、可）
  - プログラミングが面倒、難しい
  - プログラマがデータの移動を制御
  - プロセッサ数に対してスケーラブル

- 共有メモリ (shared memory)
  - 共通にアクセスできるメモリを解して、データのやり取り
  - 共有メモリシステム
  - プログラミングしやすい（逐次プログラムから）
  - システムがデータの移動を行ってくれる
  - プロセッサ数に対してスケーラブルではないことが多い。

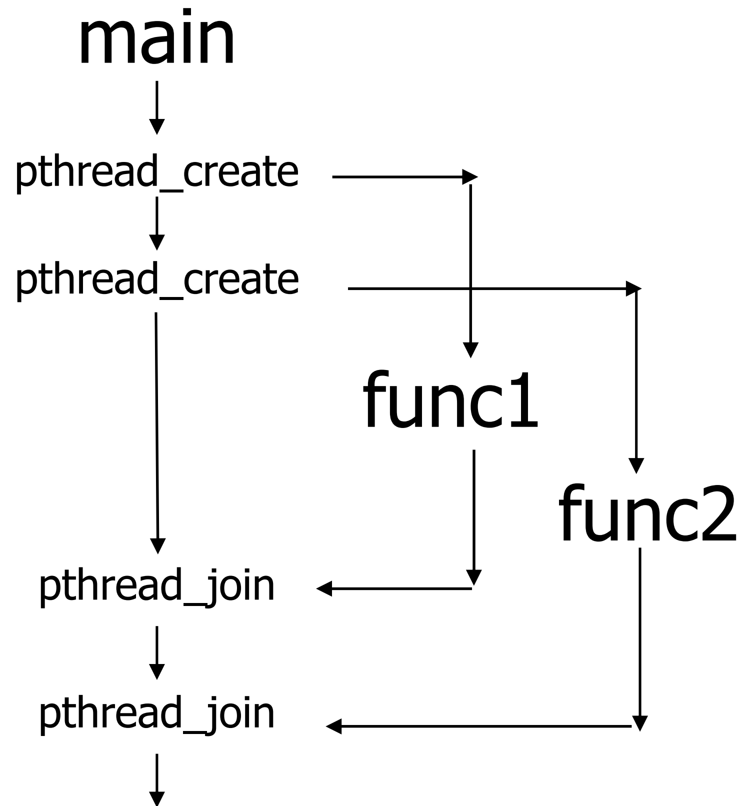
マルチスレッドプログラミング

- 共有メモリプログラミングの基礎
- スレッド：一連のプログラムの実行を抽象化したもの
  - プロセスとの違い
    - プロセスは、スレッド＋メモリ空間（メモリプロテクション）
  - POSIXスレッド pthread



POSIX threadライブラリ

- スレッドの生成 thread_create
- スレッドのjoin pthread_join
- 同期, ロック




```
#include <pthread.h>

void func1( int x );   void func2( int x );

main() {
        pthread_t t1 ;
        pthread_t t2 ;
            pthread_create( &t1, NULL,
                        (void *)func1, (void *)1 );
            pthread_create( &t2, NULL,
                        (void *)func2, (void *)2 );
            printf("main()¥n");
            pthread_join( t1, NULL );
            pthread_join( t2, NULL );
}
void func1( int x ) {
      int i ;
       for( i = 0 ; i<3 ; i++ ) {
              printf("func1( %d ): %d ¥n",x, i );
         }
}
void func2( int x ) {
              printf("func2( %d ): %d ¥n",x);
}
```

POSIXスレッドによるプログラミング

- スレッドの生成

Pthread, Solaris thread

```
for(t=1;t<n_thd;t++){
  r=pthread_create(thd_main,t)
}
thd_main(0);
for(t=1; t<n_thd;t++)
       pthread_join();
```

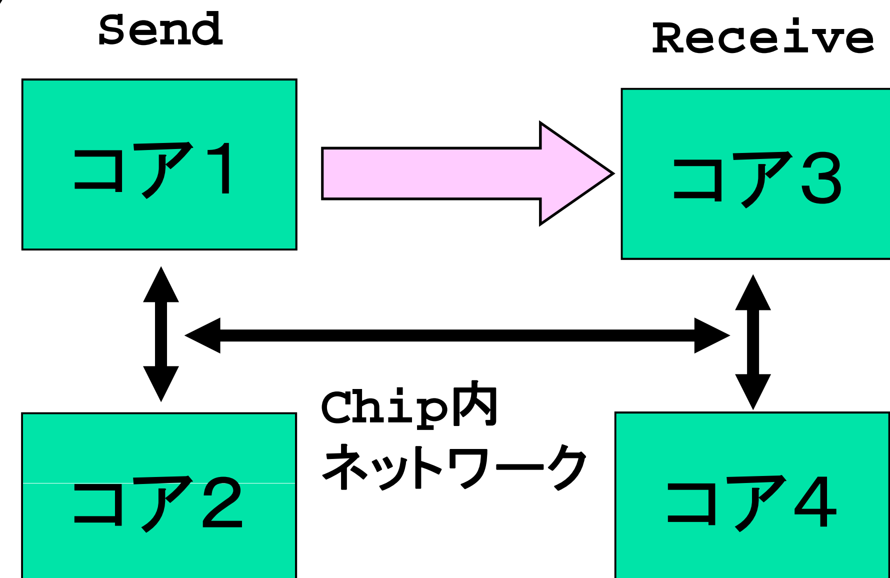
スレッド＝
プログラム実行の流れ

- ループの担当部分の分割
- 足し合わせの同期

```
int s;  /* global */
int n_thd; /* number of threads */
int thd_main(int id)
{ int c,b,e,i,ss;
  c=1000/n_thd;
  b=c*id;
  e=s+c;
  ss=0;
  for(i=b; i<e; i++) ss += a[i];
  pthread_lock();
  s += ss;
  pthread_unlock();
  return s;
}
```

メッセージ通信プログラミング

- 分散メモリの一般的なプログラミングパラダイム
  - sendとreceiveでデータ交換をする
- 通信ライブラリ・レイヤ
  - POSIX IPC, socket
  - TIPC (Transparent Interprocess Communication)
  - LINX (on Enea's OSE Operating System)
  - MCAPI (Multicore Communication API)
  - MPI (Message Passing Interface)



メッセージ通信プログラミング

- 1000個のデータの加算の例

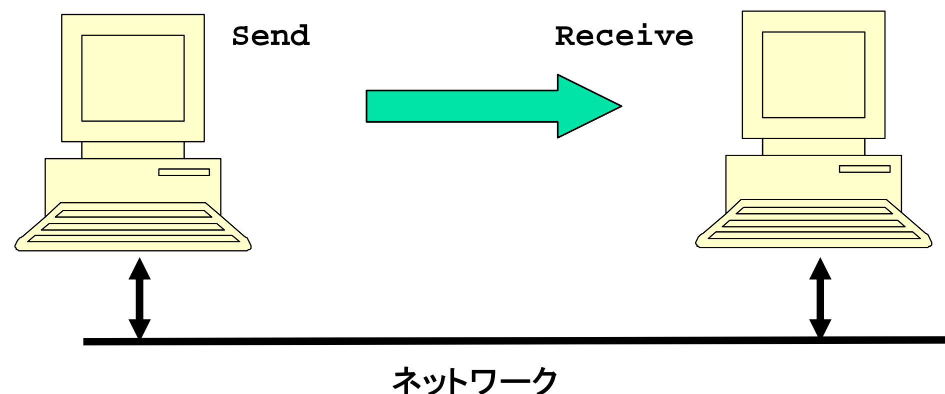
```
int a[250]; /* それぞれ、250個づづデータを持つ */

main(){     /* それぞれのプロセッサで実行される */
    int i,s,ss;
    s=0;
    for(i=0; i<250;i++) s+= a[i]; /*各プロセッサで計算*/
    if(myid == 0){    /* プロセッサ0の場合 */
        for(proc=1;proc<4; proc++){
            recv(&ss,proc);   /*各プロセッサからデータを受け取る*/
            s+=ss;    /*集計する*/
        }
    } else {   /* 0以外のプロセッサの場合 */
        send(s,0);   /* プロセッサ0にデータを送る */
    }
}
```

MPIによるプログラミング

- MPI (Message Passing Interface)
- おもに用途は、高性能科学技術計算
- 現在、ハイエンドの分散メモリシステムにおける標準的なプログラミングライブラリ
  - 100ノード以上では必須
  - 面倒だが、性能は出る
    - アセンブラでプログラミングと同じ

組み込みシステムのプログラミングには「牛刀」か！？

- メッセージをやり取りして通信を行う
  - Send/Receive
- 集団通信もある
  - Reduce/Bcast
  - Gather/Scatter



MPIでプログラミングしてみると

```
#include "mpi.h"
#include <stdio.h>
#define MY_TAG 100
double A[1000/N_PE];
int main( int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double sum, x;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d on %s¥n", myid, processor_name);

    ....
```

MPIでプログラミングしてみると

```
    sum = 0.0;
    for (i = 0; i < 1000/N_PE; i++){
       sum+ = A[i];
    }

    if(myid == 0){
       for(i = 1; i < numprocs; i++){
          MPI_Recv(&t,1,MPI_DOUBLE,i,MY_TAG,MPI_COMM_WORLD,&statu
           sum += t;
        }
    } else
           MPI_Send(&t,1,MPI_DOUBLE,0,MY_TAG,MPI_COMM_WORLD);
    /* MPI_Reduce(&sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM
    MPI_Barrier(MPI_COMM_WORLD);
    ...
    MPI_Finalize();
    return 0;
}
```

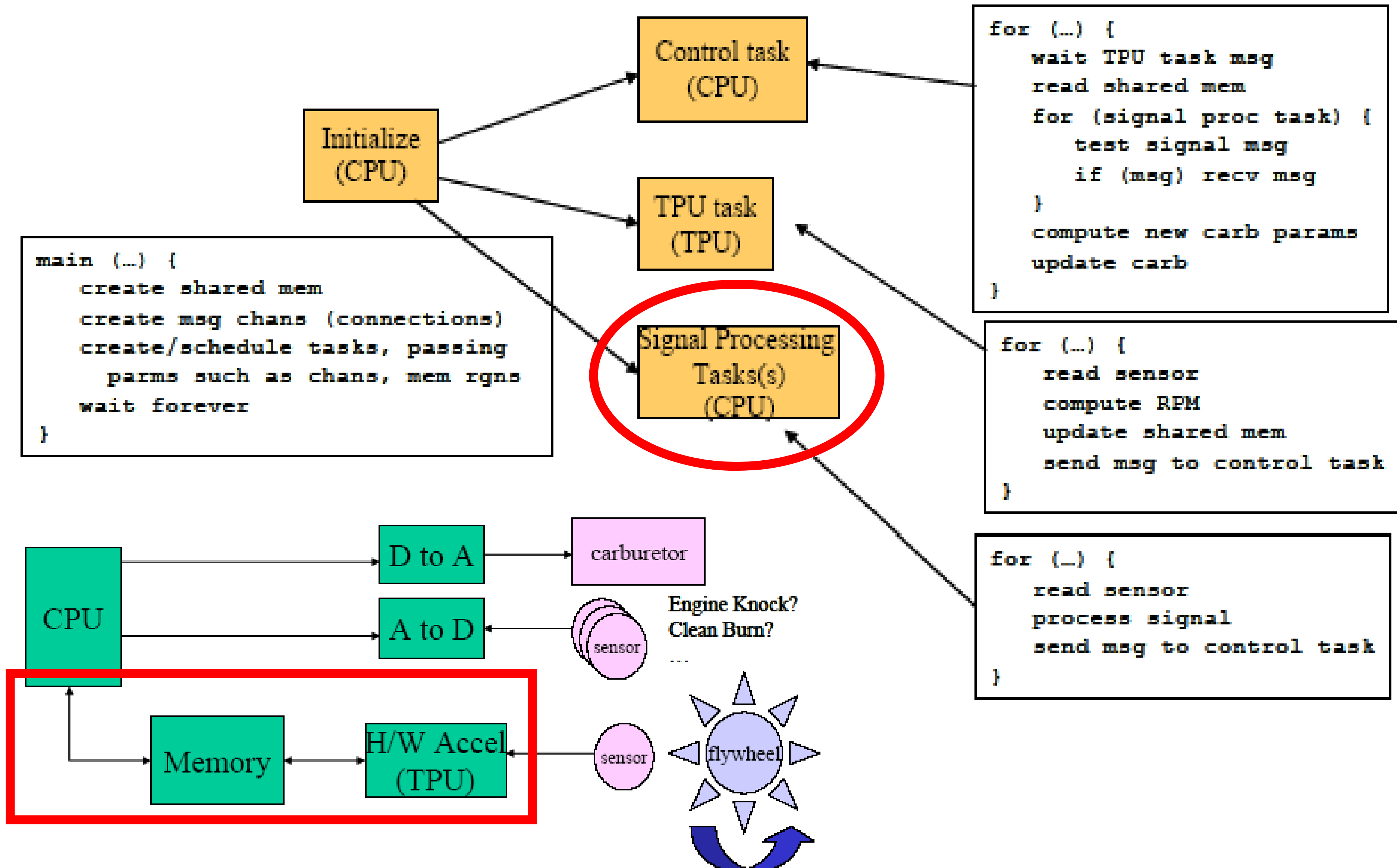
MCAPI

- MCAPI (Multicore Communication API)
  - Multicore Association (www.multicore-association.org, Intel, Freescale, TI, NEC)で制定された通信API
  - March 31, 2008 時点でV1.063
  - MRAPI (Resource Management API)ともに用いる
  - MPIよりも簡単、hetero, scalable, fault tolerance(?), general

- 3つの基本的な機能
  - 1. Messages – connection-less datagrams.
  - 2. Packet channels – connection-oriented, uni-directional, FIFO packet streams.
  - 3. Scalar channels – connection-oriented single word uni-directional, FIFO packet streams.

- MCAPI's objective is to provide a limited number of calls with sufficient communication functionality while keeping it simple enough to allow efficient implementations.

例



```
//////////////////////////////////////////
// The TPU task
//////////////////////////////////////////
void TPU_Task() {
        char* sMem;
        size_t msgSize;
        mcapi_endpoint_t cntrl_endpt, cntrl_remote_endpt;
        mcapi_sclchan_send_hndl_t cntrl_chan;
        mcapi_request_t r1;
        mcapi_status_t err;

        // init the system
        mcapi_initialize(TPU_NODE, &err);
        CHECK_STATUS(err);
        cntrl_endpt =
        mcapi_create_endpoint(TPU_PORT_CNTRL, &err);
        CHECK_STATUS(err);
        mcapi_get_endpoint_i(CNTRL_NODE,
        CNTRL_PORT_TPU,
        &cntrl_remote_endpoint, &r1, &err);
        CHECK_STATUS(err);

        // wait on the remote endpoint
        mcapi_wait(&r1,NULL,&err);
        CHECK_STATUS(err);

        // now get the shared mem ptr
        mcapi_msg_recv(cntrl_endpt, &sMem,
        sizeof(sMem), &msgSize, &err);
        CHECK_MEM(sMem);
        CHECK_STATUS(err);

        // NOTE – connection handled by control task
        // open the channel
        mcapi_open_sclchan_send_i(&cntrl_chan,
        cntrl_endpt, &r1, &err);
        CHECK_STATUS(err);
        // wait on the open
        mcapi_wait(&r1,NULL,&err);
        CHECK_STATUS(err);

        // ALL bootstrapping is finished, begin  processing
        while (1) {
                // do something that updates shared mem
                sMem[0] = 1;
                // send a scalar flag to cntrl process
                // indicating sMem has been updated
                mcapi_sclchan_send_uint8(cntrl_chan,
                (uint8_t) 1,&err);
                CHECK_STATUS(err);
        }
}
```

OpenMPとは

- 共有メモリマルチプロセッサの並列プログラミングのためのプログラミングモデル
  - 言語ではない。
  - ベース言語(Fortran/C/C++)をdirective（指示文）で並列プログラミングできるように拡張
  - もともとは、（いまでも）科学技術計算向け
  - マルチコアプロセッサの普及とともに、共有メモリの性能向上のためのプログラミングモデルとして注目されている

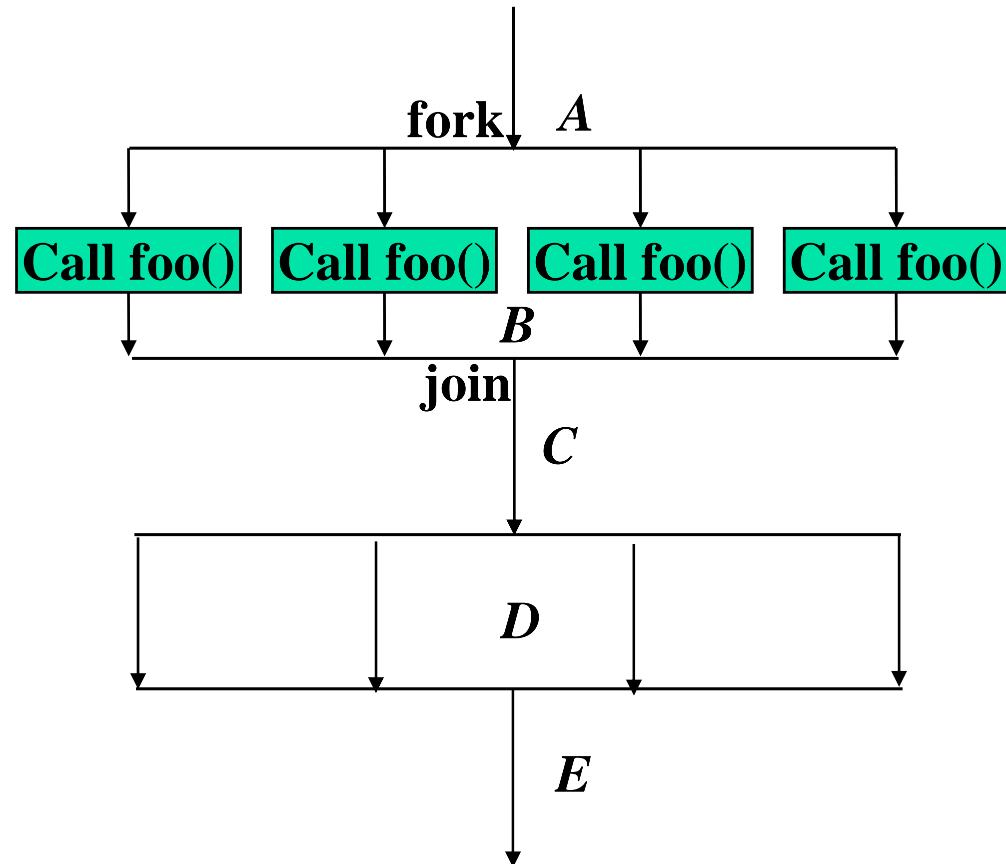
- OpenMP Architecture Review Board (ARB)が仕様を決定
  - 当初、米国コンパイラ関係のISVを中心に組織
  - Oct. 1997 Fortran ver.1.0 API
  - Oct. 1998 C/C++ ver.1.0 API
  - 現在、OpenMP 3.0

- http://www.openmp.org/

OpenMPの実行モデル

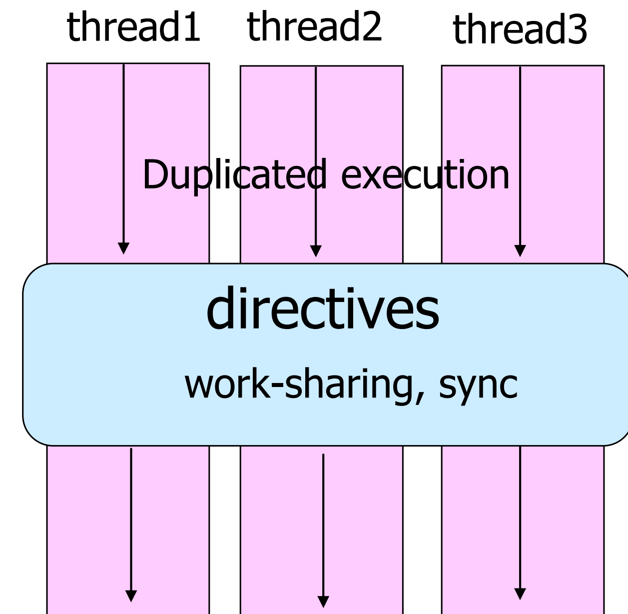
- 逐次実行から始まる
- Fork-joinモデル
- parallel region
  - 関数呼び出しも重複実行

```
… A …
#pragma omp parallel
{
    foo(); /* ..B… */
}
… C ….
#pragma omp parallel
{
… D …
}
… E …
```



Work sharing構文

- Team内のスレッドで分担して実行する部分を指定
  - parallel region内で用いる
  - `for` 構文
    - イタレーションを分担して実行
    - データ並列
  - `sections`構文
    - 各セクションを分担して実行
    - タスク並列
  - `single`構文
    - 一つのスレッドのみが実行



  - parallel 構文と組み合わせた記法
    - `parallel for` 構文
    - `parallel sections`構文

For構文

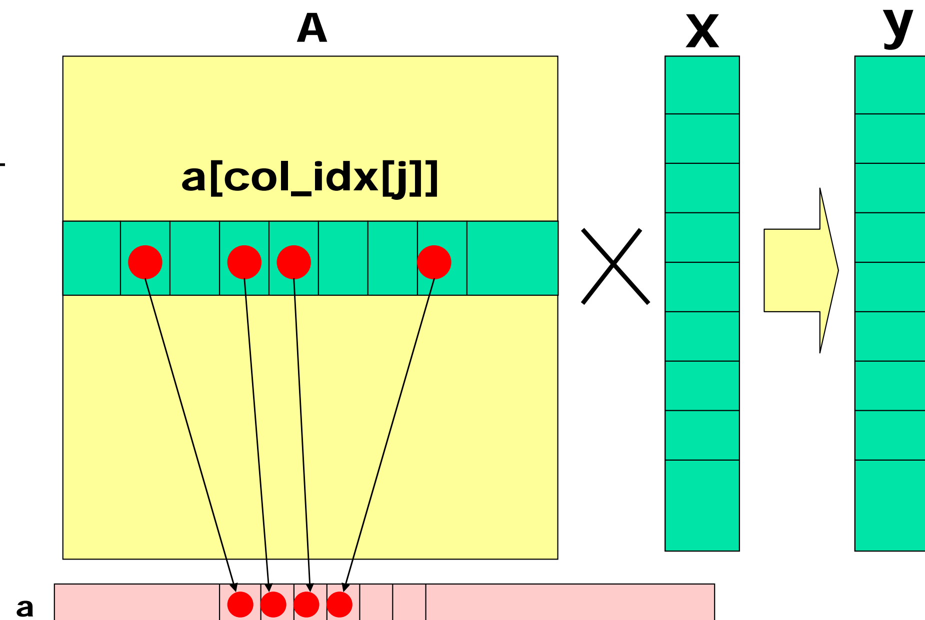
- Forループのイタレーションを並列実行
- 指示文の直後のforループは*canonical shape*でなくてはならない

```
#pragma omp for [clause…]
 for(var=lb; var logical-op ub; incr-expr)
    body
```

  - *var*は整数型のループ変数（強制的にprivate）
  - *incr-expr*
    - ++*var*, *var*++ , --*var*, *var*-- , *var*+=*incr*, *var*-=*incr*
  - *logical-op*
    - ＜、＜＝、＞、＞＝
  - ループの外の飛び出しはなし、breakもなし
  - *clause*で並列ループのスケジューリング、データ属性を指定

例　疎行列ベクトル積ルーチン

```
Matvec(double a[],int row_start,int col_idx[],
 double x[],double y[],int n)
{
   int i,j,start,end; double t;
#pragma omp parallel for private(j,t,start,end)
   for(i=0; i<n;i++){
      start=row_start[i];
      end=row_start[i+1];
      t = 0.0;
      for(j=start;j<end;j++)
         t += a[j]*x[col_idx[j]];
      y[i]=t;
  }
}
```



Data scope属性指定

- `parallel`構文、work sharing構文で指示節で指定
- `shared(var_list)`
  - 構文内で指定された変数がスレッド間で共有される
- `private(var_list)`
  - 構文内で指定された変数がprivate
- `firstprivate(var_list)`
  - privateと同様であるが、直前の値で初期化される
- `lastprivate(var_list)`
  - privateと同様であるが、構文が終了時に逐次実行された場合の最後の値を反映する
- `reduction(op:var_list)`
  - reductionアクセスをすることを指定、スカラ変数のみ
  - 実行中はprivate、構文終了後に反映

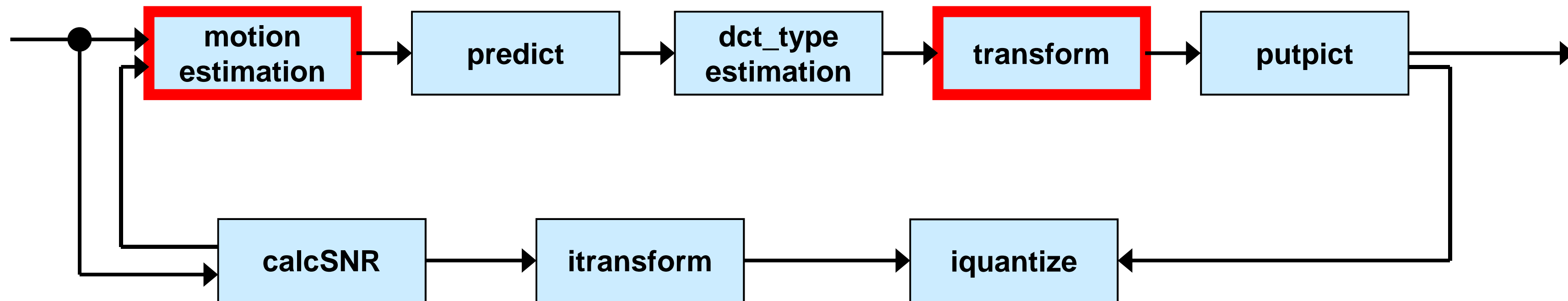
Barrier 指示文

- バリア同期を行う
  - チーム内のスレッドが同期点に達するまで、待つ
  - それまでのメモリ書き込みもflushする
  - 並列リージョンの終わり、work sharing構文でnowait指示節が指定されない限り、暗黙的にバリア同期が行われる。

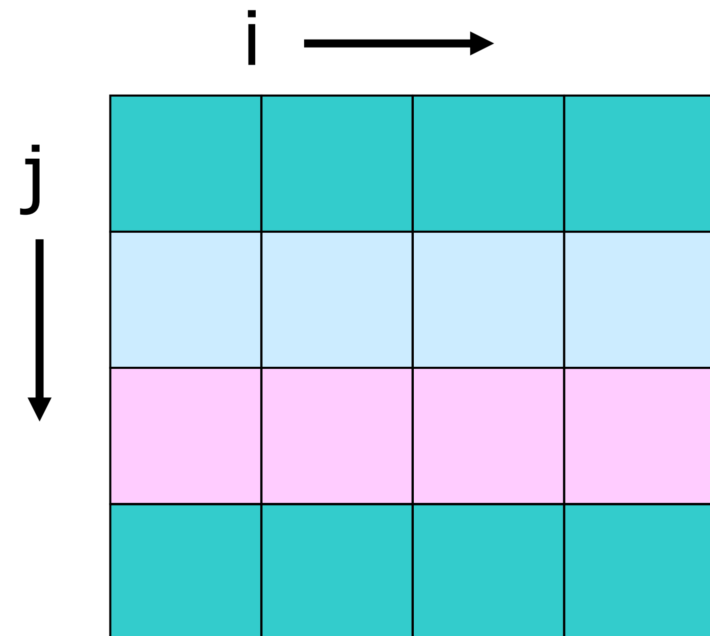
```
#pragma omp barrier
```

MediaBench

- MPEG2 encoder by OpenMP.

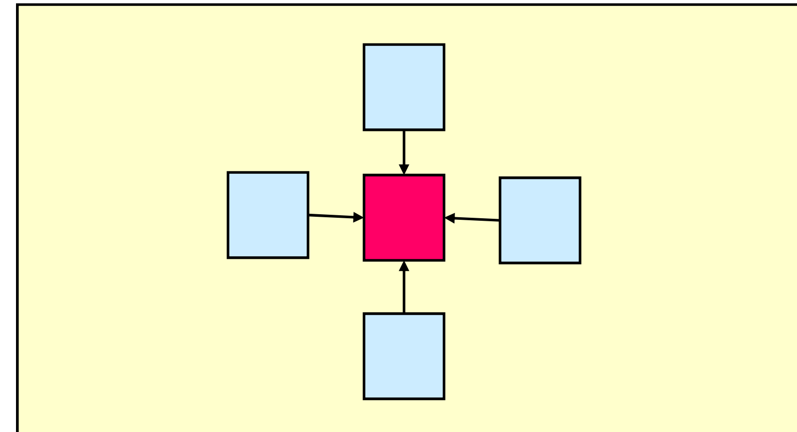



```
/*loop through all macro-blocks of the picture*/
#pragma omp parallel private(i,j,myk)
{
#pragma omp for
    for (j=0; j<height2; j+=16)
    {
        for (i=0; i<width; i+=16)
        {
            ... loop body ...
        }
    }
}
```



OpenMPのプログラム例：laplace

- Laplace方程式の陽的解法
  - 上下左右の4点の平均で、updateしていくプログラム
  - Oldとnewを用意して直前の値をコピー
  - 典型的な領域分割
  - 最後に残差をとる



- OpenMP版 lap.c
  - 3つのループを外側で並列化
    - OpenMPは1次元のみ
  - Parallel指示文とfor指示文を離してつかってみた

```
void lap_solve()
{
    int x,y,k;
    double sum;

#pragma omp parallel private(k,x,y)
{
    for(k = 0; k < NITER; k++){
         /* old <- new */
#pragma omp for
         for(x = 1; x <= XSIZE; x++)
           for(y = 1; y <= YSIZE; y++)
             uu[x][y] = u[x][y];
         /* update */
#pragma omp for
         for(x = 1; x <= XSIZE; x++)
           for(y = 1; y <= YSIZE; y++)
             u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
    }
 }

/* check sum */
    sum = 0.0;
#pragma omp parallel for private(y) reduction(+:sum)
    for(x = 1; x <= XSIZE; x++)
         for(y = 1; y <= YSIZE; y++)
           sum += (uu[x][y]-u[x][y]);
    printf("sum = %g¥n",sum);
}
```

OpenMP3.0で追加された点

www.openmp.orgに富士通訳の日本語バージョンの仕様書がある

- タスクの概念が追加された
  - Parallel 構文とTask構文で生成されるスレッドの実体
  - task構文
  - taskwait構文
- メモリモデルの明確化
  - Flushの扱い
- ネストされた場合の定義の明確化
  - Collapse指示節
- スレッドのスタックサイズの指定
- C＋＋でのprivate変数に対するconstructor, destructorの扱い

Task構文の例

```
struct node {
        struct node *left;
        struct node *right;
};

void postorder_traverse( struct node *p ) {
        if (p->left)
                #pragma omp task // p is firstprivate by default
                postorder_traverse(p->left);
        if (p->right)
                #pragma omp task // p is firstprivate by default
                postorder_traverse(p->right);
        #pragma omp taskwait
        process(p);
}
```

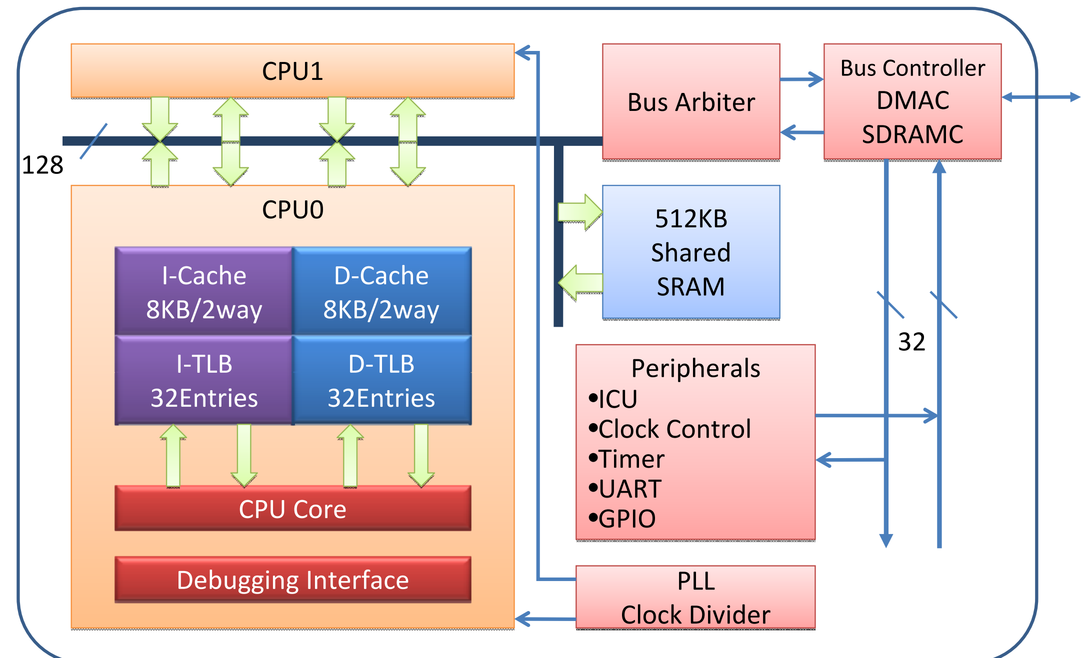
M32700の特徴

ルネサステクノロジ (旧三菱電機)

- M32R-II コア x 2
  - 7段パイプライン
  - 32bit命令(1命令同時発行+16bit命令(2命令同時発行可能)
  - 浮動小数点ユニットは持たない
    - gcc付属の浮動小数点ライブラリ (soft-float)

- 内蔵512KB SRAM
  - 今回は未使用
- SDRAMコントローラ内蔵

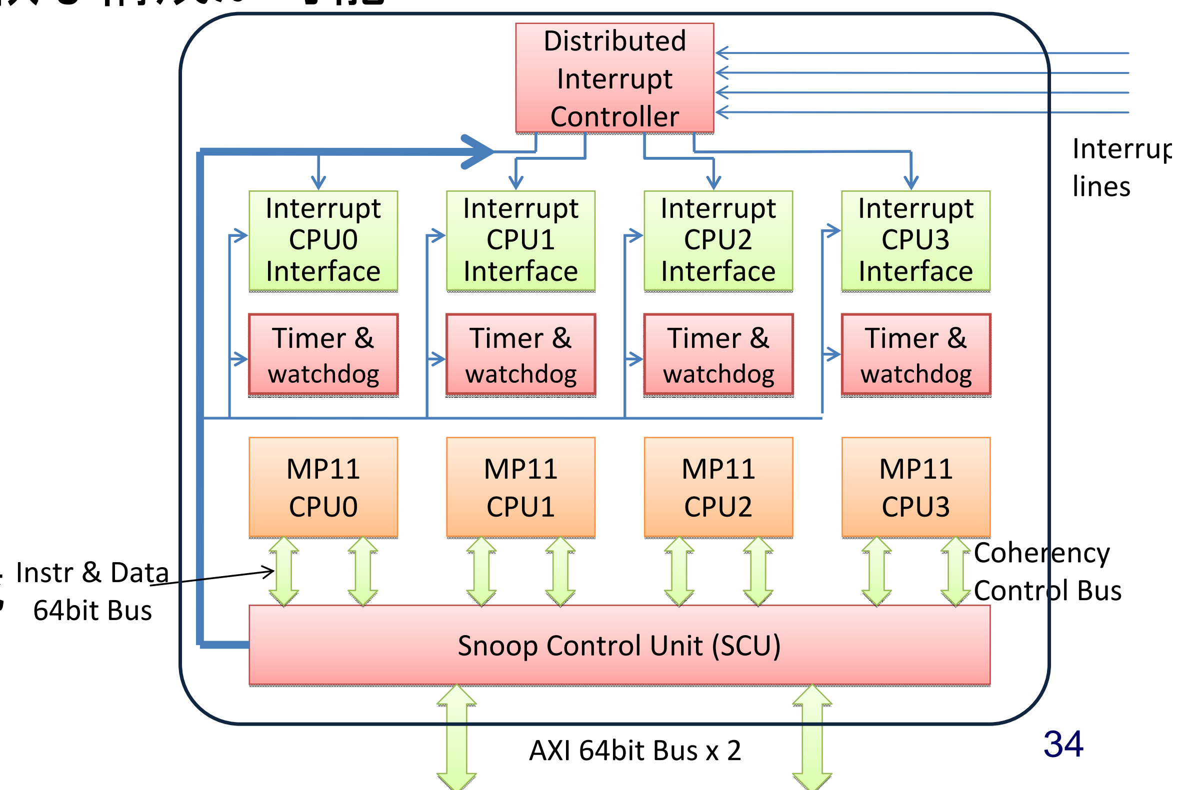
- μT-Engine
  M3T-32700UTを使用



MPCoreの特徴

ARM+NECエレクトロニクス

- ARM MP11コア(ARM11アーキテクチャ) x 4
  - ARMv6命令セット、ARM命令セット(32bit), Thumb命令セット(16bit), Jazelle命令セット(可変長)
  - 8段パイプライン、1命令同時発行
  - チップ実装の際には柔軟な構成が可能
  - L2 cache, 1MB, 8way-set-assoc



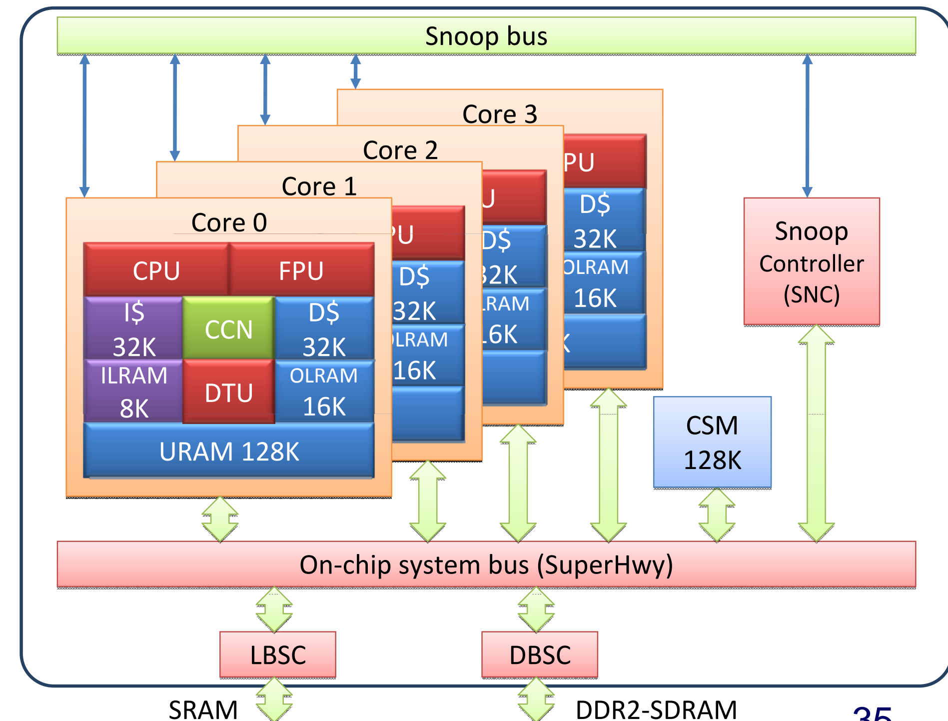
- CT11MPCore + RealView Emulation Baseboardを使用
  - DDR-SDRAMコントローラなど周辺I/FはFPGAに搭載

RP1の特徴

早稲田大+ルネサステクノロジ+日立

- SH-X3アーキテクチャ，SH-4Aコア x 4
  - 16bit命令セット，2命令同時発行可能
  - 8段パイプライン
- 専用のスヌープバス
  - SHwyのトラフィックを避けて転送
- オンチップメモリ…

  今回は未使用
  - ローカルメモリ
    - 命令用 ILRAM (8Kbyte, 1clock)
    - データ用 OLRAM (8Kbyte, 1clock)
    - URAM (128Kbyte, 1～数クロック)
  - 集中共有メモリ (CSM, 128Kbyte)



各プロセッサの比較

	ルネサス M32700	ARM+NEC MPCore	早大+ルネサス+日立 RP1	Intel Core2Quad Q6600
コア数	2	4	4	4
コア周波数	300MHz	210MHz	600MHz	2.4GHz
内部バス周波数	75MHz	210MHz	300MHz	
外部バス周波数	75MHz	30MHz	50MHz	
キャッシュ(I+D)	2way 8K+8K	4way 32K+32K L2, 1MB, 8way	4way 32K+32K	8way 32K+32K (L1) 16way 4M(2コア) x 2 (L2)
ラインサイズ	16byte	32byte	32byte	64byte
主記憶	32MB SDRAM 100MHz	256MB DDR-SDRAM 30MHz	128MB DDR2-600 300MHz	4GB DDR2-800 400MHz

実行時ライブラリの実装

- Omni OpenMPコンパイラでは、排他制御を行なう関数の選択が可能
  - POSIX thread のmutex lock
  - 特定アーキテクチャ専用のスピンロック実装
- 今回用いた各マルチコアプロセッサ向けに，スピンロックを用いた実行時ライブラリを実装
  - `SIMPLE_SPIN`をdefine
    - 同期操作を簡略化、特に parallelディレクティブの軽量化
    - （複数プロセスの場合に影響が出る可能性あり）
  - ロック変数をFalse sharingを防ぐためキャッシュラインサイズでalign

同期性能の評価

NPTLでもスピンロックの方がMPCoreで1.1倍程度高速
⇒ 以後スピンロックのみを採用

- EPCC マイクロベンチ
  - Edinburgh Parallel Computi

Linuxthreadsではスピンロックの方が
M32700で最大482倍、RP1で最大695倍高速

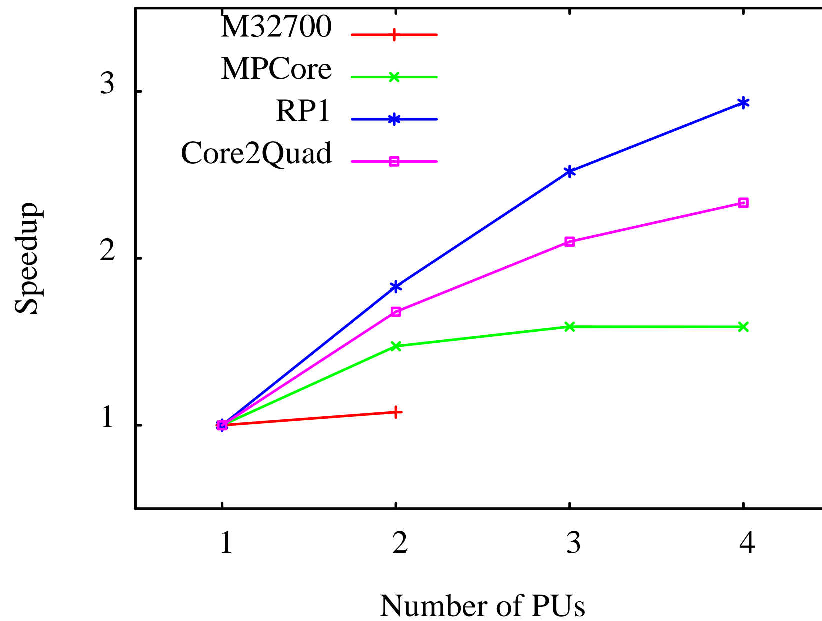
Parallelディレクティブ関連では
メモリアクセス速度の影響が大きい

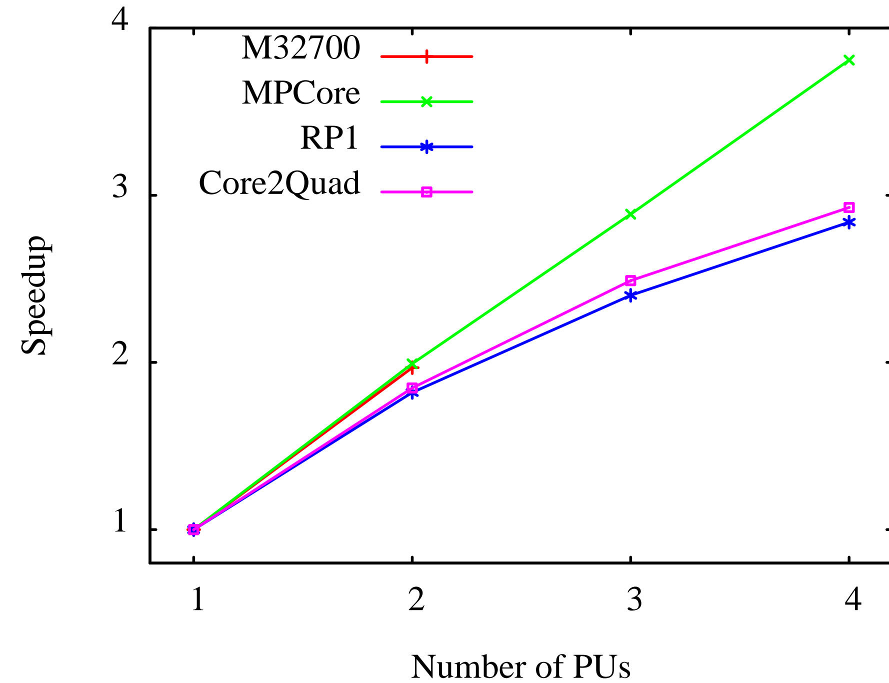
チマーク

上段：Mutex，下段：スピンロック，単位：μ秒

	parallel	for	parallel for	barrier	single	critical	lock/unlock	ordered	atomic	reduction
M32700	392.2	18.5	399.7	14.1	50.8	273.5	273.1	8.64	241.0	401.9
	376.8	13.6	383.6	10.7	9.87	3.15	2.51	7.08	0.501	387.1
MPCore	436.5	7.46	436.3	6.11	3.14	0.921	1.03	1.50	0.894	443.9
	434.8	6.15	435.7	5.98	3.12	0.837	0.962	1.33	0.893	443.8
RP1	107.8	1.66	108.2	1.13	295.1	128.2	121.0	0.584	121.0	327.0
	107.2	1.42	107.7	0.867	1.53	0.190	0.174	0.598	0.365	109.1
Q6600	2.80	0.364	3.71	0.301	4.54	1.31	1.41	0.191	0.474	6.13
	2.25	0.372	2.47	0.316	0.859	0.129	0.131	0.168	0.307	3.35

NPB IS, CGの結果

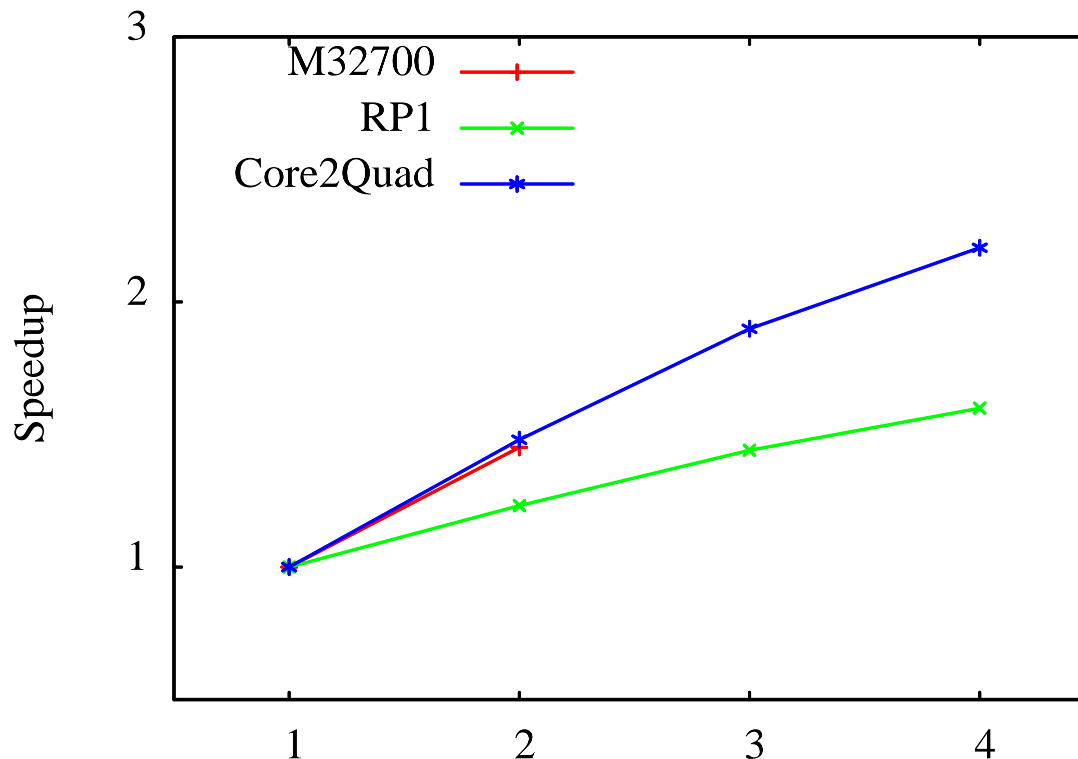




メモリインテンシブ、
メモリアクセス性能の影響
が大きい

キャッシュにヒットしやすい
計算インテンシブ

Mpeg2encの結果

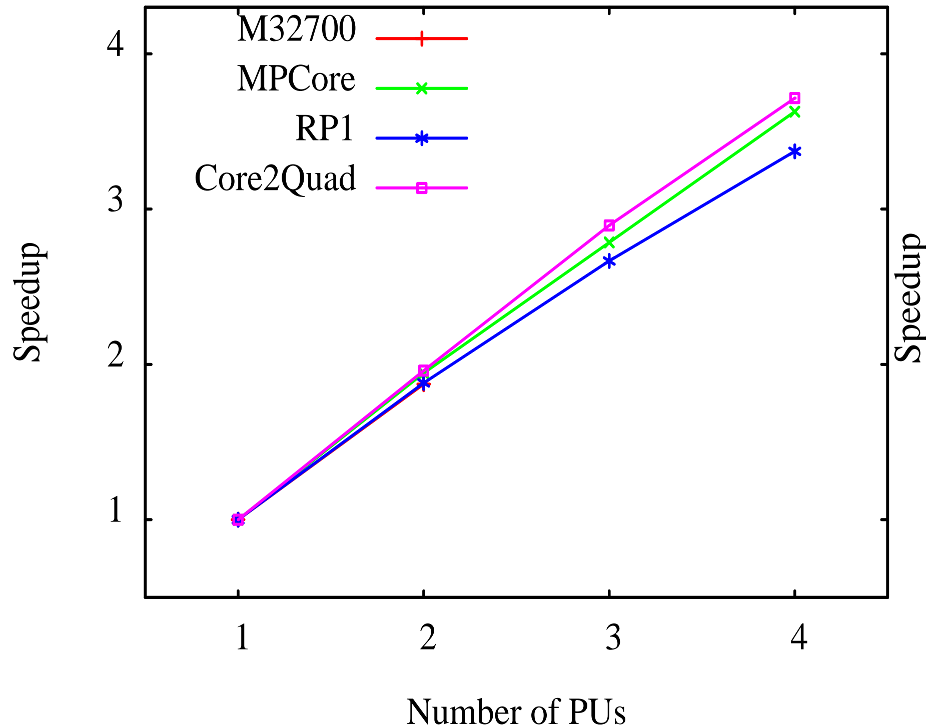


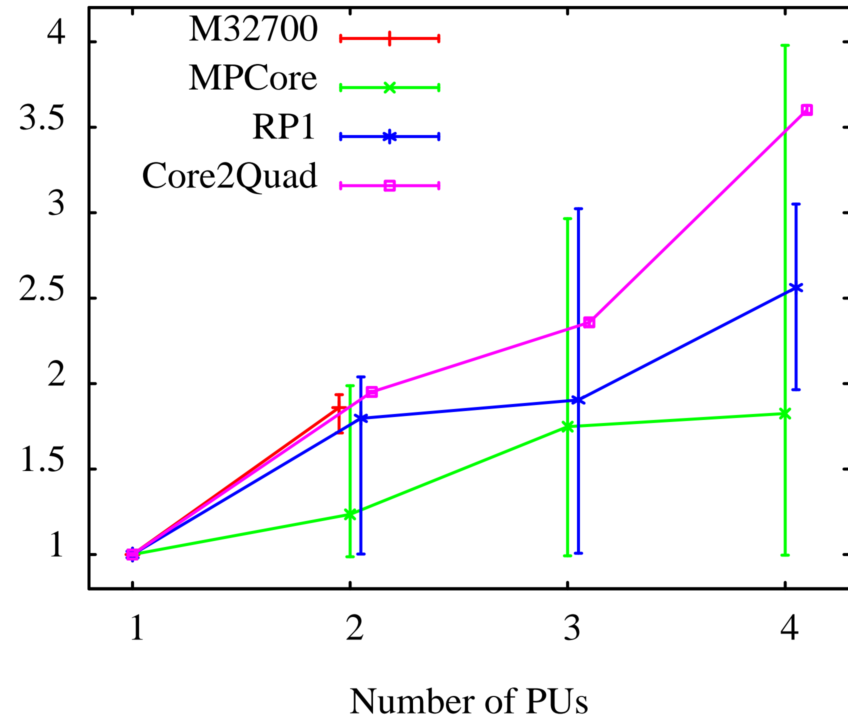
MPCoreでは実行エラー

- ファイル入力を伴う → NFSの影響
- M32700では浮動小数点演算の ソフトウェアエミュレーションが隠ぺい

Susan smoothing 、BlowFish (ECBモード)の結果

ばらつきが大きいため、error barで表
- error bar: 最大値と最小値、折れ線: 平均値

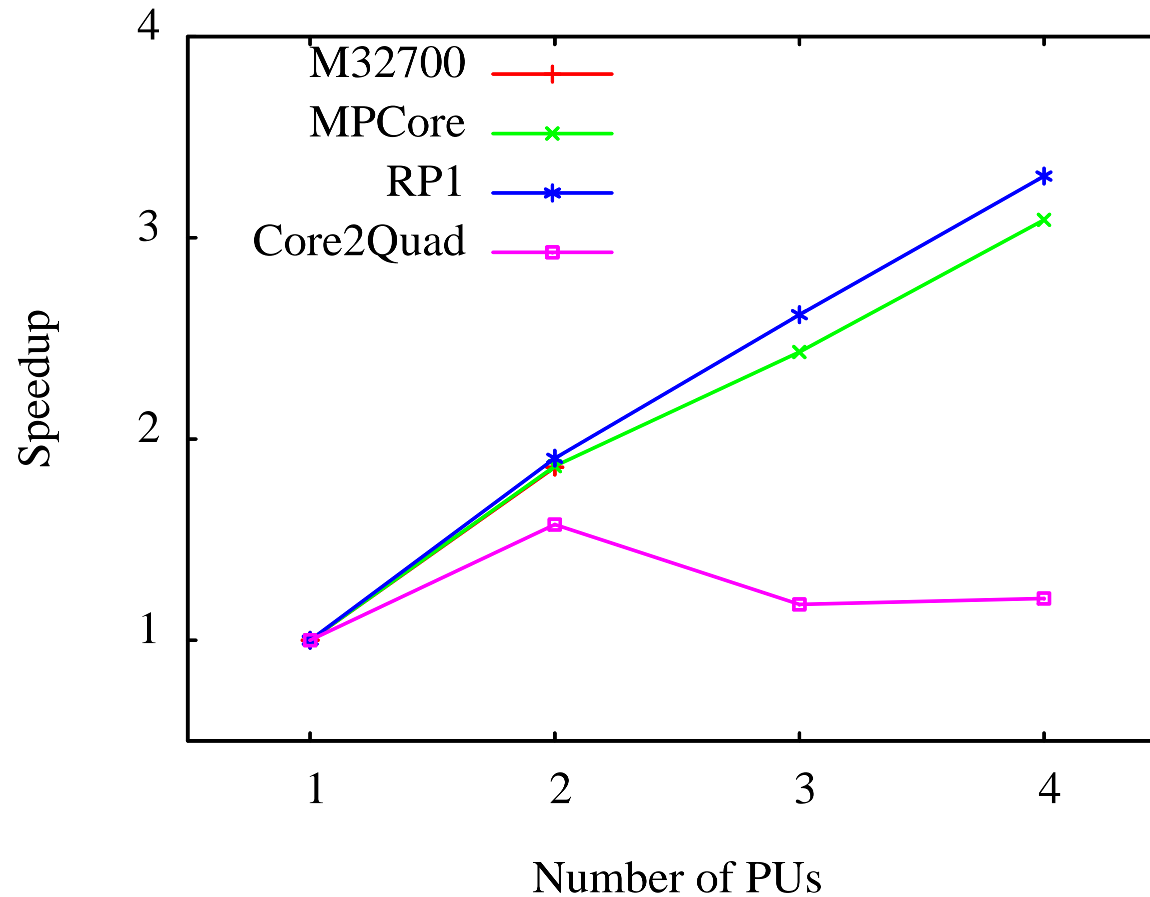




高い性能向上

ブロックの処理毎にファイル入出力を伴う
Core2Quad以外はNFS環境

FFTの結果



Core2Quadは実行時間が極めて短い（数ミリ秒）

⇒ オーバヘッドの方が大きい

OpenMP化のコスト

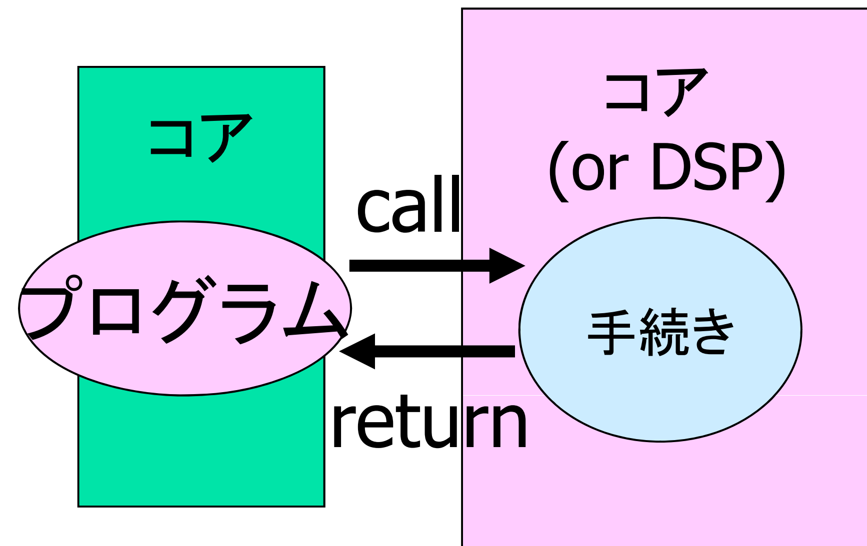
方針

- parallelディレクティブ(fork-join)のコストが大きいので、なるべくparallelリージョンをまとめて長くする

アプリケーション	変更行数
susan smoothing	ディレクティブ6行追加
Blowfish encoding	ディレクティブ9行追加 12行修正
FFT	ディレクティブ4行追加
Mpeg2enc	ディレクティブ5行追加 7行修正

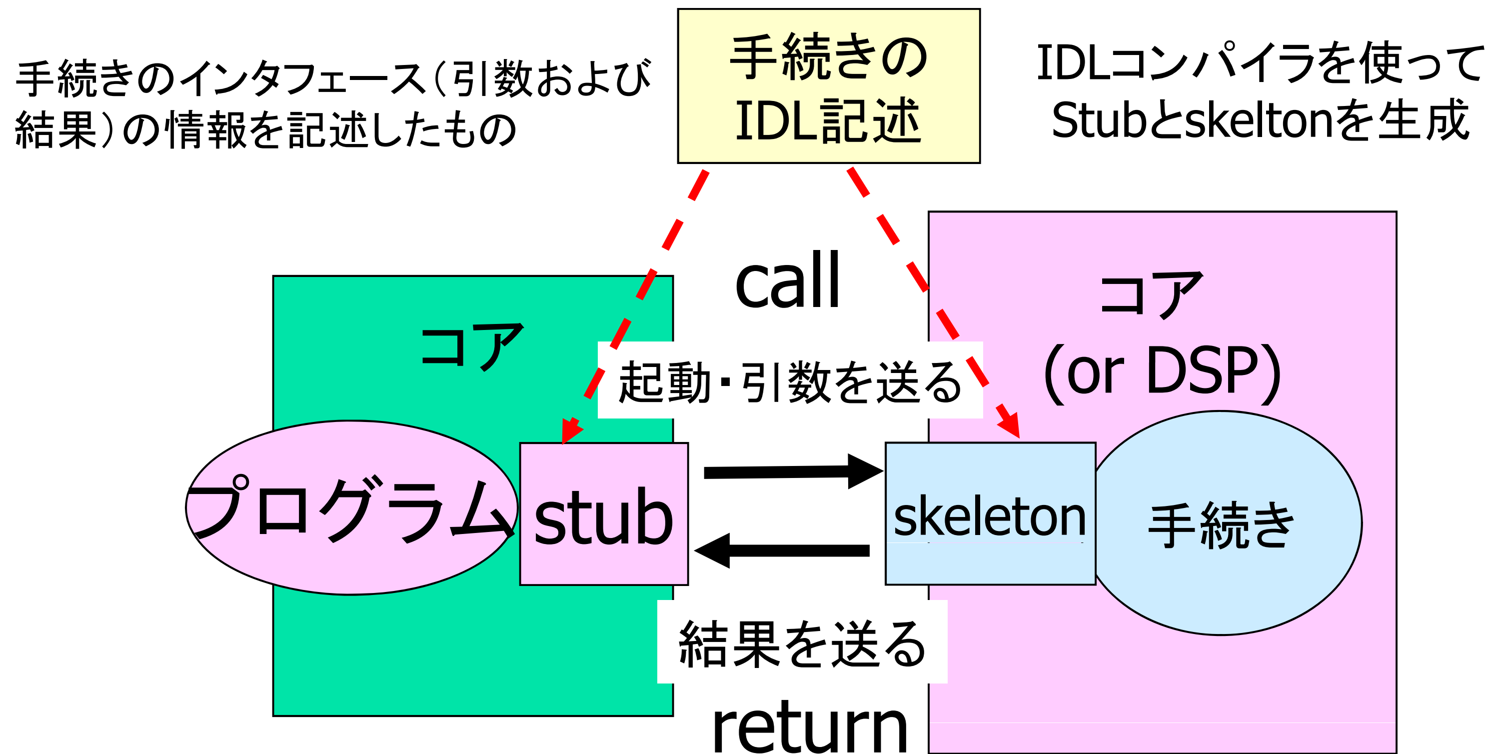
RPCによるマルチコアプログラミング

- RPC (remote procedure call)
  - 別のアドレス空間（通常、共有ネットワーク上の別のコンピュータ上）にある手続きを実行することを可能にする技術
  - client-server(caller-callee)に抽象化し、通信の詳細を隠蔽
    - IDL (interface description language)でインタフェースを記述、通信を生成
  - いろいろな分野・実装・応用がある
    - SUN RPC – システムサービス
    - CORBA (common object broker arch)
    - GridRPC
- マルチコアでも使える
  - 既存のルーチンを違うコアに割り当てる
  - AMPの形態には自然な抽象化
    - “ある機能を呼び出す”
    - もちろん、SMPでもOK
  - 通信を隠蔽してくれるので、
    分散メモリ、共有メモリどちらでもよい



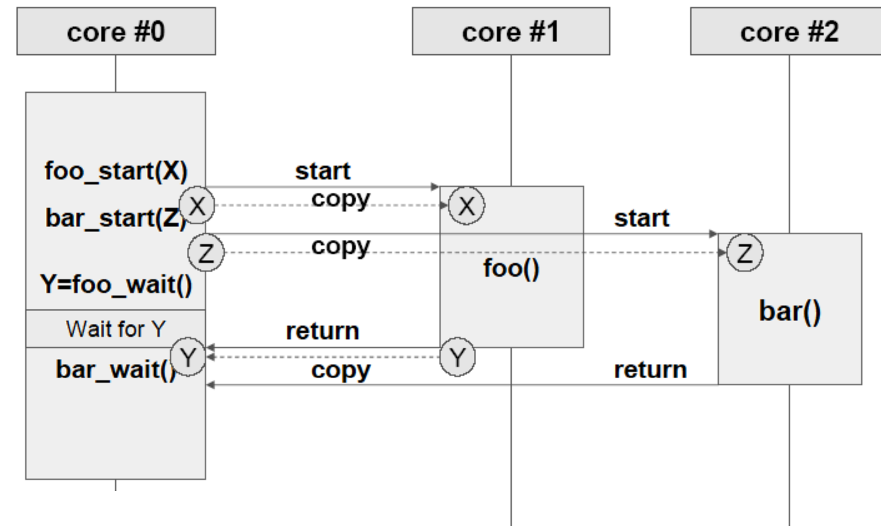
RPCの仕組み

- client-server(caller-callee)に抽象化し、通信の詳細を隠蔽
  - IDL (interface description language)でインタフェースを記述、通信を生成
  - Stub - クライアント側のメソッド呼び出しをサーバにディスパッチ
  - Skeleton - サーバ側でクライアントに代わってメソッドを呼び出す



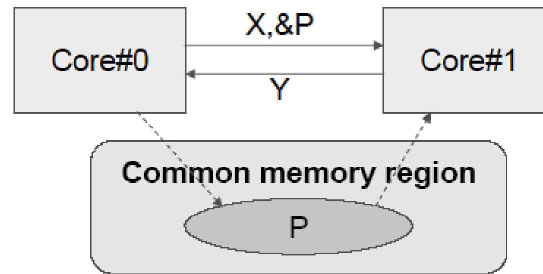
富士通・非同期RPC(ARPC)によるマルチコアプログラミング

- 富士通が非同期RPC(ARPC)によるマルチコアプログラミングを提案
- 非同期＝複数のRPCを同時に実行



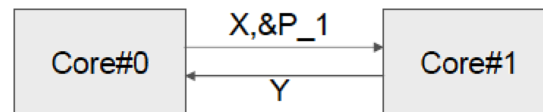
Common memory region

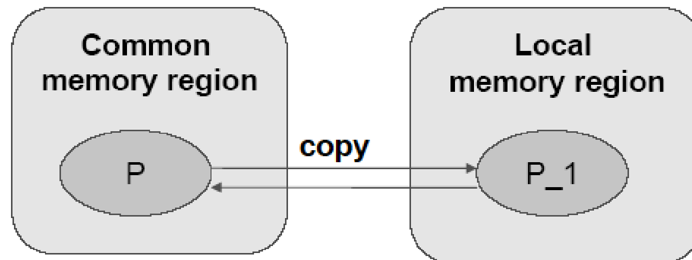
```
foo_start(&handle, X, &P);

…                 } Inhibit  access to P

Y=foo_wait(&handle);
```



Local memory region

```
move_start(&handle1,
           &P_1, &P, size);
move_wait(&handle1);
foo_start(&handle2, X, &P_1);

…

Y=foo_wait(&handle2);
move_start(&handle1,
           &P, &P_1, size);
move_wait(&handle1);
```

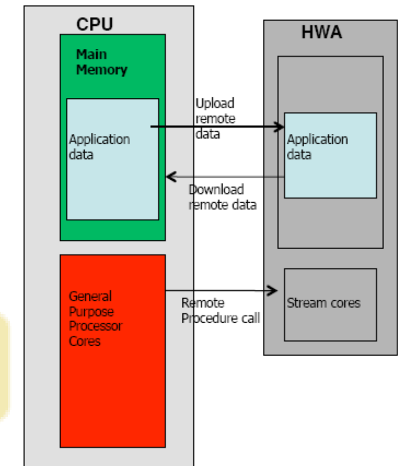




- 通常の逐次プログラムからの移行が簡単
- 通信を隠蔽することにより、いろいろなコア(&DSP)に対して移植性の良いプログラムができる
  - ⇒ 開発コストの低減

RPCによるマルチコアプログラミングのこれから

- 逐次プログラムからの移行が容易, いろいろな形態(AMP&SMP、DSP)に対応
- directiveベースのプログラミング環境も提案されている
  - HMPP (hybrid multicore parallel programming)@INRIA
  - StarSs @BSC




```
#include <stdio.h>
#include <stdlib.h>

#pragma hmpp simple codelet, args[1].io=out
void simplefunc(int n, float v1[n], float v2[n], float v3[n], float alpha)
{
  int i;
  for (i = 0 ; i< n ; i++) {
    v1[i] = v2[i] * v3[i] + alpha;
  }
}
int main(int argc, char **argv) {
  unsigned int n = 400;
  float t1[400], t2[400], t3[400];
  float alpha = 1.56;
  unsigned int  j, seed = 2;
/* Initialization of input data*/
/* . . . */

#pragma hmpp simple callsite
  simplefunc(n,t1,t2,t3,alpha);

  printf("%f %f (...) %f %f \n", t1[0], t1[1], t1[n-2], t1[n-1]);
  return 0;
}
```

codelet / callsite
directive set

組み込みマルチコアプロセッサのプログラミングの課題

- 標準化（の不在）
  - 組み込みプロセッサ・システムは形態が多様
  - Chip内のインターコネクトの通信ソフトウエア
    - MCAPI (Multicore Communication API)は本命？
  - 標準的な（高レベルな）プログラミングモデル、簡便なプログラミング環境
    - ARPC ? OpenMP?
  - いずれにしろ、分散メモリになる？だから、…
- 実時間処理と並列処理
  - （特に共有メモリSMPの場合）並列のプロセスのスケジューリングが、実時間処理と相性が悪い？！
  - 実時間処理の場合、必要な資源（コア）がすぐに割り当てられないといけない
  - コアを意識したスレッドの割り当て (core affinity)
    - Linux2.6からsched_setaffinityがあるが、HPCにはOKだが、組み込みには不十分
- デバック…