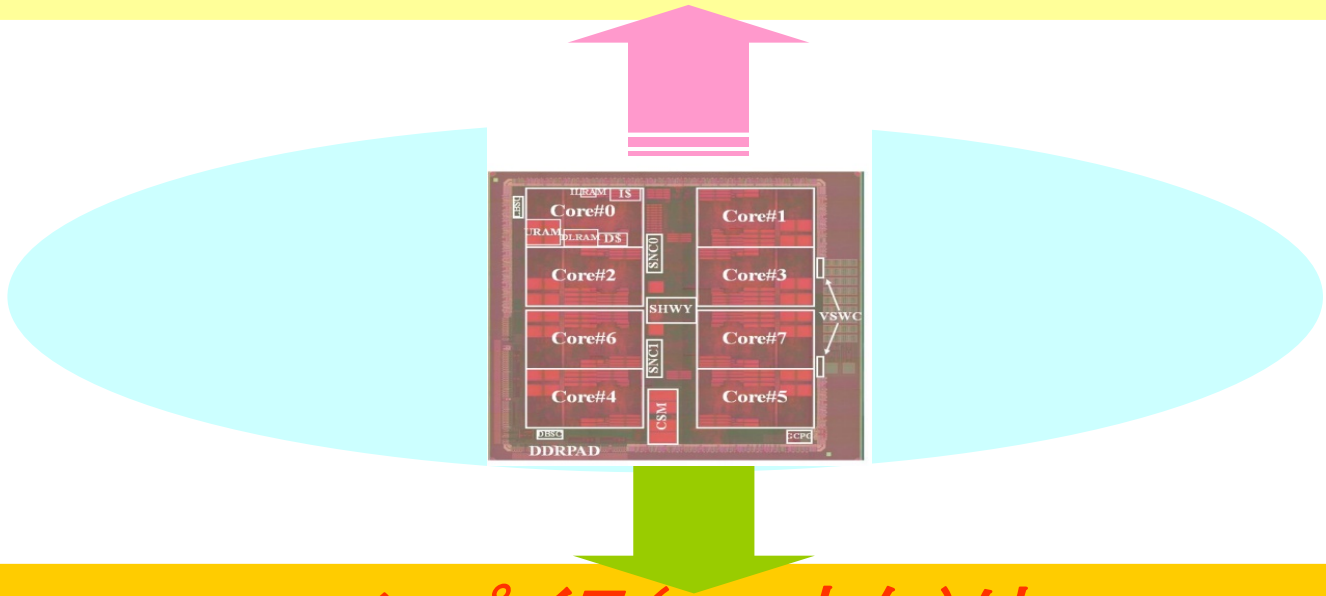
マルチコア用コンパイル技術の現在

早稲田大学 情報理工学科 木村啓二

マルチコア用ソフトウェア開発

- デスクトップはもちろん、HPC、組み込みにも普及
- 限られた実装コストで高い性能を得るため
- 限られた電力で高い性能を得るため



コンパイラ(gccとか)は
マルチコア用の最適化をしてくれるの？

そもそもマルチコアの
アーキテクチャとは？

複数コアを使うための
方法とは？

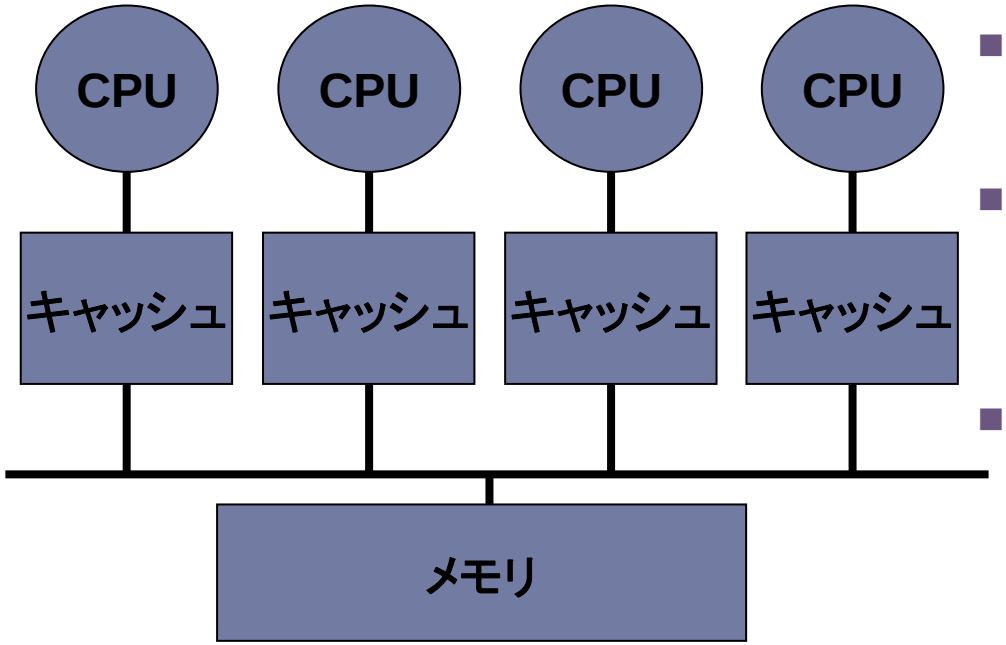
これまでのコンパイラや
その技術は？

最後にOSCAR
コンパイラを紹介

マルチコアプロセッサの構成例

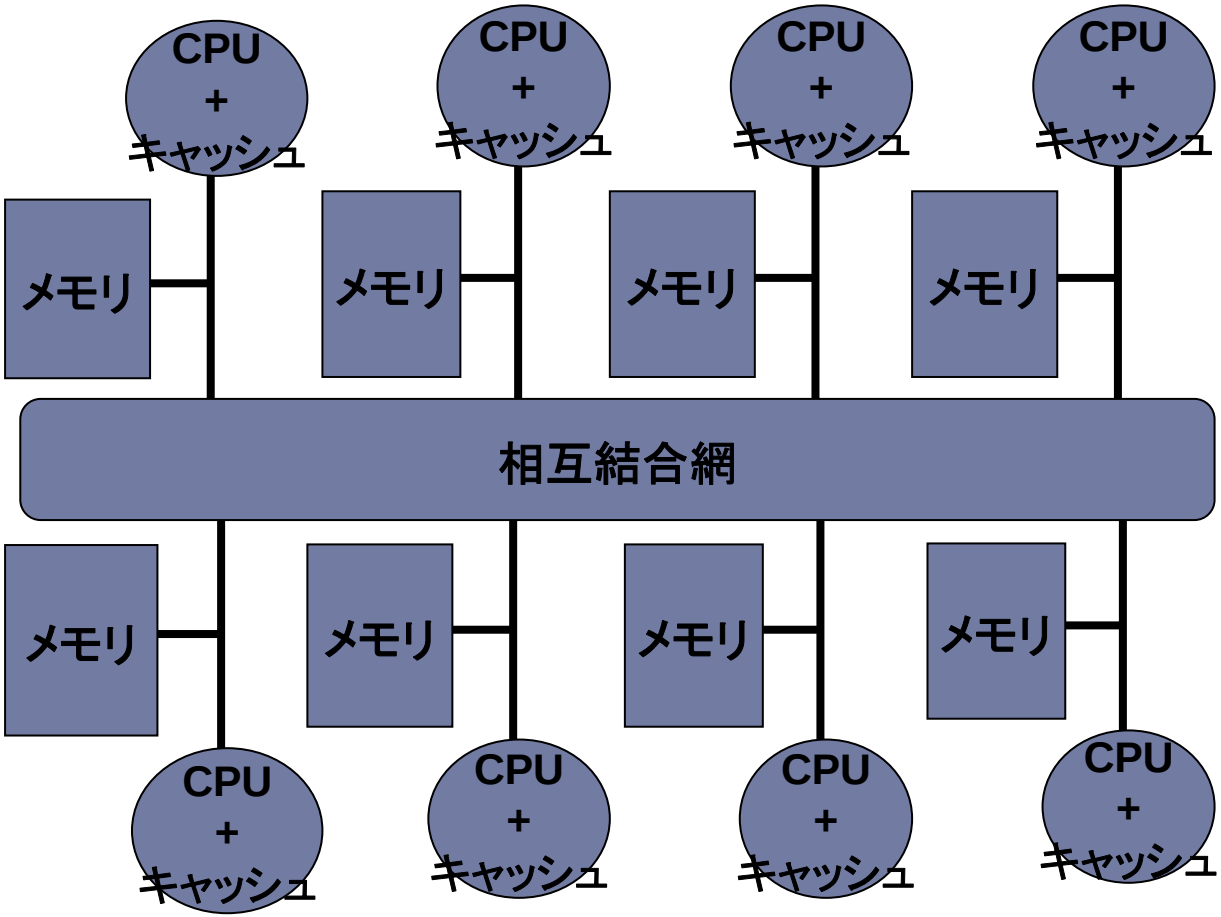
- コアのつなぎ方からみた基本的な構成例
  - Symmetric Multi Processor (SMP)型
    - メモリを共有
  - 分散メモリ型
    - コア（CPU）独自のメモリを持つ

マルチコアプロセッサの構成例
SMP型



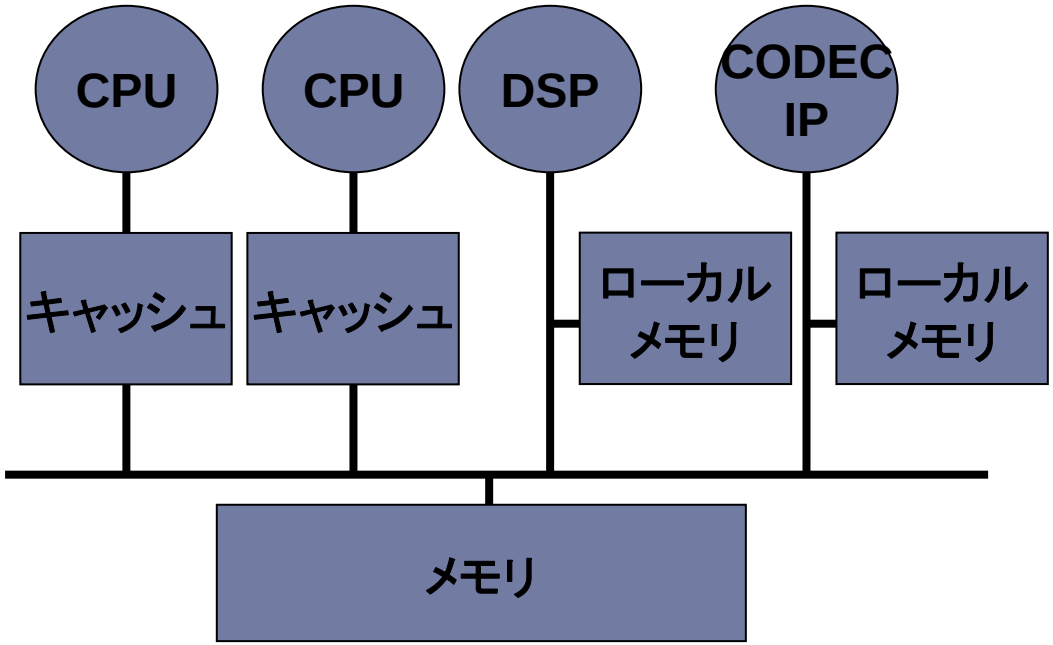
- 全てのCPUがメモリを共有
- キャッシュ間で一貫性を保証する必要がある
- 小規模のマルチプロセッサで一般的
  - CPU数を増やしにくい
- Core Duoなど，キャッシュを共有する構成もある

マルチコアプロセッサの構成例
分散メモリ型



- メモリは各CPUに分散
- 明示的なデータ転送が必要
- 大規模構成をとりやすい

マルチコアプロセッサの構成例
ヘテロジニアスな構成



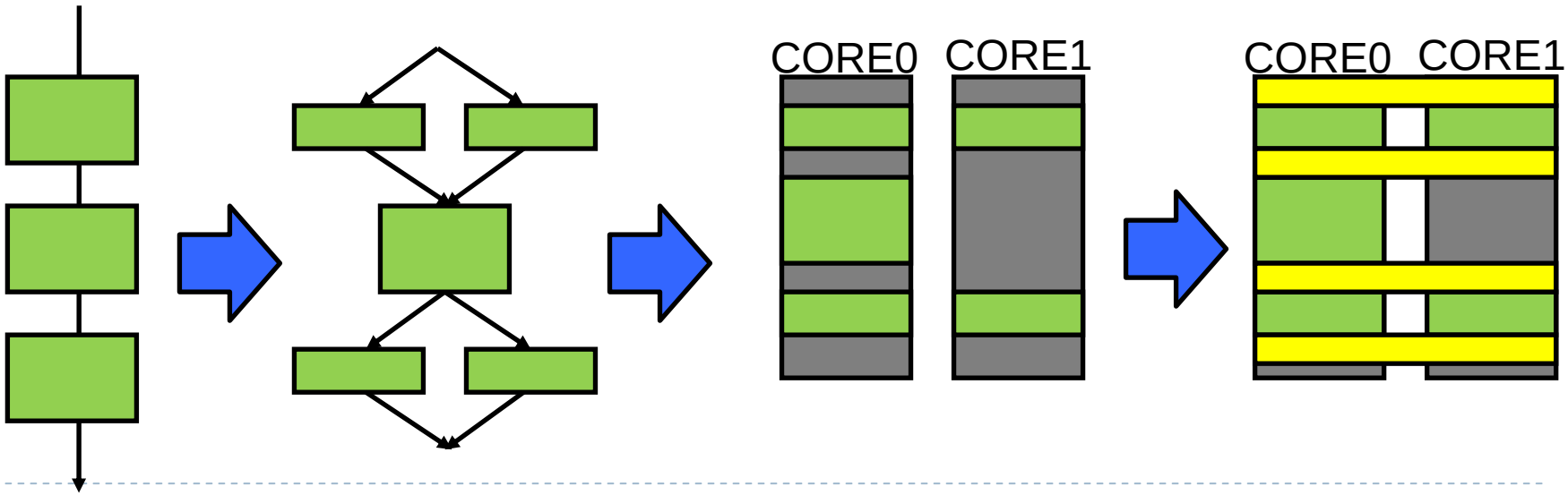
- 組み込み形では割とよく見る構成
- 異種な計算資源を持つ
- ターゲットアプリケーションを意識した構成

マルチコアをどのように使うか

- 機能分散
  - 各コアに独立した機能を割り当てる
- 並列処理
  - 一つのプログラムを複数のコアで処理する
- そもそも動機は？
  - 必要な処理性能の確保
  - 低消費電力化
- ではプログラムはどうやって書くか？
  - 普通のCを使う？
    - コンパイラが並列処理のことをやってくれる？
  - プログラムに並列処理の記述を追加する？
    - まずはこちらの説明から
    - 後の方で最近の話題も

プログラムの並列化とは

- 並列化可能な箇所の特定
- 並列処理単位(タスク)への分割
- タスクのコアへの割り当て(スケジューリング)
- 同期コード・(必要なら)データ転送コードの挿入



pthread（参考）

- UNIX系OSで標準的なスレッドライブラリ
  - Cygwinでも利用可能
- 各スレッド（プログラム中の処理単位）がメモリを共有しているモデル
  - SMP型を前提としている
  - スタック以外のデータをスレッド間で共有
- コンパイラは関係ない

MPI（参考）

- 分散メモリ型アーキテクチャ向けのAPI
- プロセス間のデータ転送を記述する
- C, FORTRANが対象
  - 出自はhigh performance computing
- コンパイラは関係ない

MPI 並列化の例

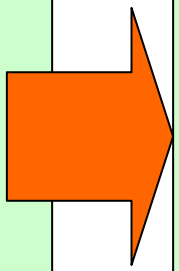
```
int
main(int argc, char** argv)
{
  int  i, j, k;
  double r;
  static double a[SIZE][SIZE];
  static double b[SIZE][SIZE];
  static double c[SIZE][SIZE];

  int myrank, nprocs;
  int isize;
  MPI_Status  status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  isize = SIZE / nprocs;

  if (myrank == 0) {
    for (i = 1; i < nprocs; i++) {
      MPI_Send(&a[i*isize][0], isize*SIZE, MPI_DOUBLE,
                       i, FROM0, MPI_COMM_WORLD);
      MPI_Send(b,              SIZE*SIZE,  MPI_DOUBLE,
                       i, FROM0, MPI_COMM_WORLD);
    }
  } else {
    MPI_Recv(a, isize*SIZE, MPI_DOUBLE, 0, FROM0,
                MPI_COMM_WORLD, &status);
    MPI_Recv(b, SIZE*SIZE,  MPI_DOUBLE, 0, FROM0,
                MPI_COMM_WORLD, &status);
  }

  for (i = 0; i < isize; i++) {
    for (j = 0; j < SIZE; j++) {
      r = 0;
      for (k = 0; k < SIZE; k++) {
        r += a[i][k]*b[k][j];
      }
      c[i][j] = r;
    }
  }

  if (myrank == 0) {
    for (i = 1; i < nprocs; i++) {
      MPI_Recv(&c[i*isize][0], isize*SIZE, MPI_DOUBLE, i, TO0,
                       MPI_COMM_WORLD, &status);
    }
  } else {
    MPI_Send(c, isize*SIZE, MPI_DOUBLE, 0, TO0,
                     MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

OpenMP

- C, C++, FORTRANに対する並列処理記述用API
  - こちらも出自はhigh performance computing
- 並列処理を行いたい場所にコンパイラ指示文を挿入
- SMPを前提としたメモリモデル
- Intelのコンパイラなど多数のコンパイラがサポート
  - gcc 4.2でもサポート

OpenMP
並列化の例

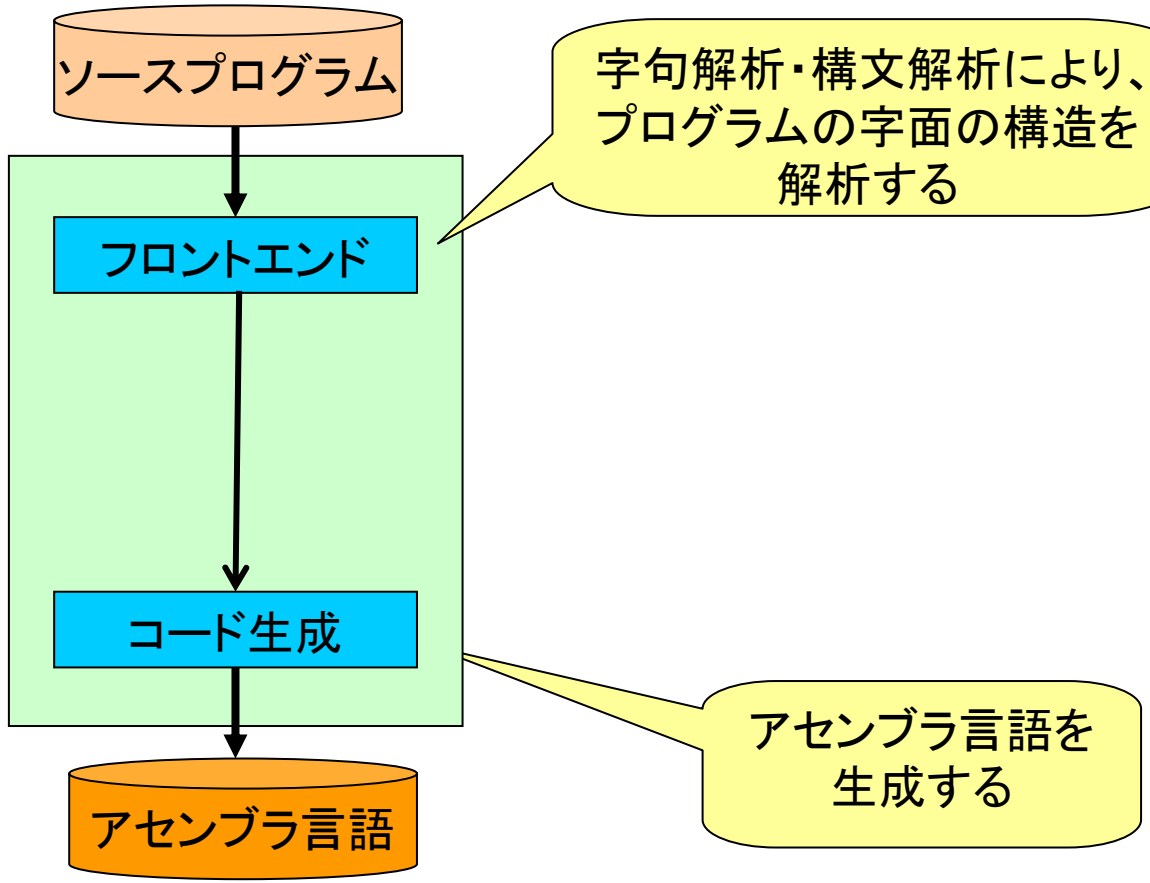
```
int
main(int argc, char** argv)
{
  int  i, j, k;
  double r;
  static double a[SIZE][SIZE];
  static double b[SIZE][SIZE];
  static double c[SIZE][SIZE];

  for (i=0; i<SIZE; i++) {
    for (j=0; j<SIZE; j++) {
      r=0;
      for (k=0; k<SIZE; k++) {
        r+=a[i][k]*b[k][j];
      }
      c[i][j]=r;
    }
  }

  return 0;
}
```



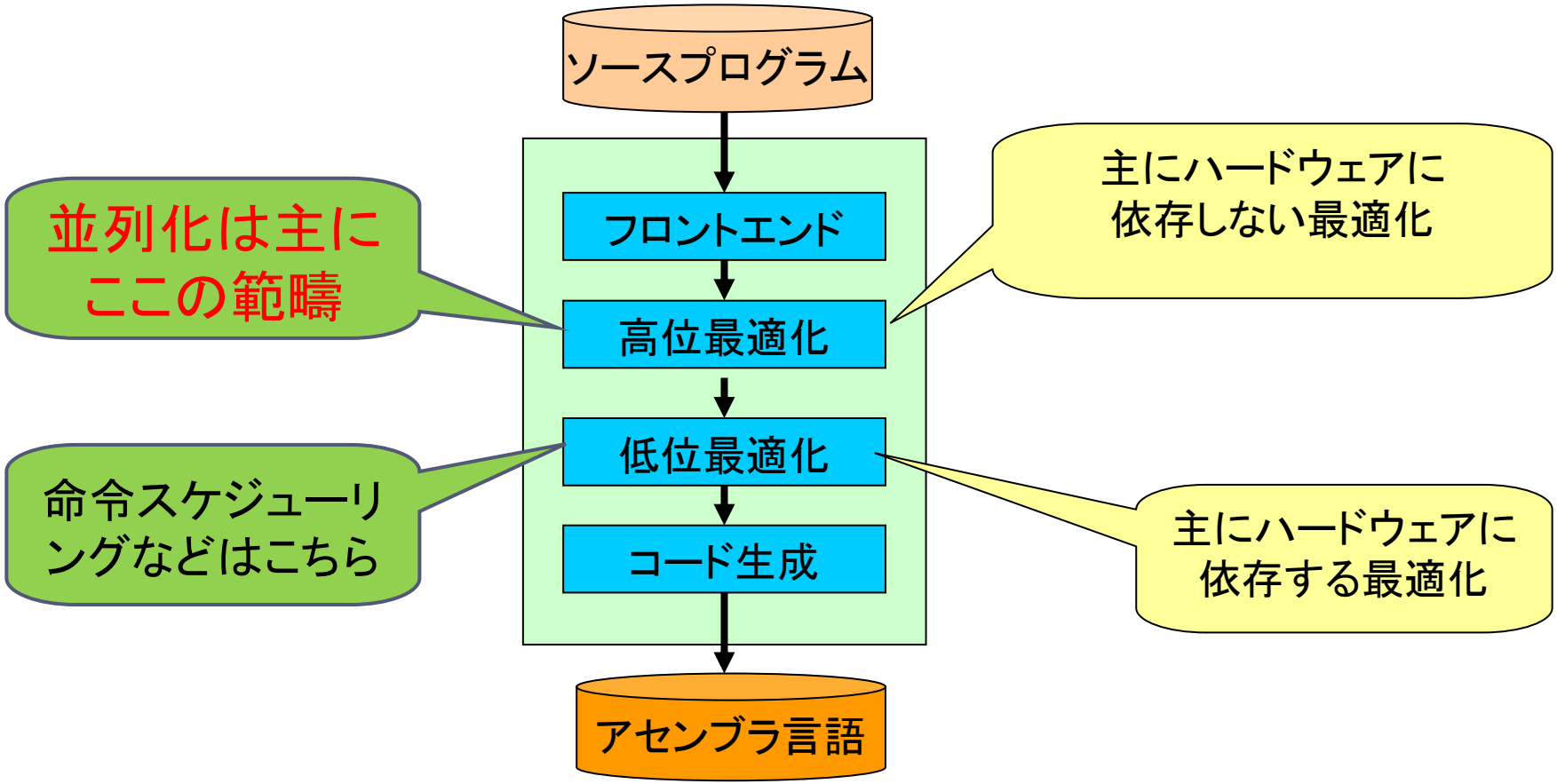
```
int
main(int argc, char** argv)
{
  int  i, j, k;
  double r;
  static double a[SIZE][SIZE];
  static double b[SIZE][SIZE];
  static double c[SIZE][SIZE];

#pragma omp parallel for private(i,j,k,r)
  for (i=0; i<SIZE; i++) {
    for (j=0; j<SIZE; j++) {
      r=0;
      for (k=0; k<SIZE; k++) {
        r+=a[i][k]*b[k][j];
      }
      c[i][j]=r;
    }
  }

  return 0;
}
```

コンパイラの構成（おさらい）

学部ぐらいの
授業内容



コンパイラの構成（もう少し細かく）



並列処理が可能な条件とは？

- 処理の順番を変えても計算結果が変わらない
- 計算結果が変わってしまう条件とは？
  - ある処理の結果を後ろの処理が使用する
  - ある処理が使うデータを後ろの処理が上書きする
  - ある処理の結果を後ろの処理が上書きする

これらを「二つの処理の間に依存がある」という

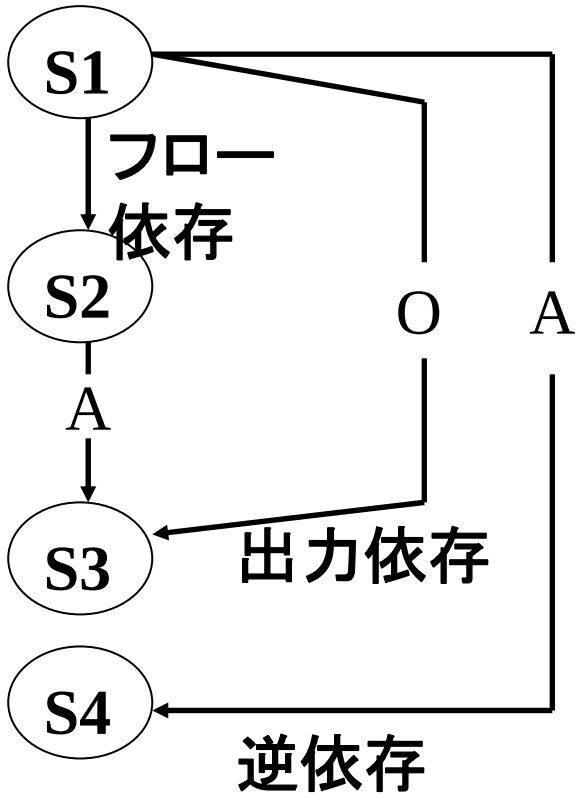
依存の種類

A=B+C : S1

D=A+E : S2

A=C+F : S3

B=E+F : S4



マルチプロセッサ向けの並列化: ループ並列化

- あるループを複数の処理に小分けして各プロセッサに割り当てる
  - ループ内部の各繰り返しが同時実行できるかどうか解析する

```
for (i=0; i<1000; i++)
  sum += a[i];
```

CPU0用

```
for (i=0; i<250; i++)
  sum0 += a[i];
barrier(var); /*待ち合わせ*/
sum = sum0+sum1+sum2+sum3;
```

CPU1用

```
for (i=250; i<500; i++)
  sum1 += a[i];
barrier(var);
```

CPU2用

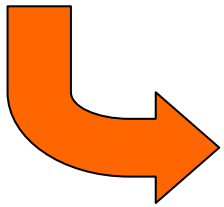
```
for (i=500; i<750; i++)
  sum2 += a[i];
barrier(var);
```

CPU3用

```
for (i=750; i<1000; i++)
  sum3 += a[i];
barrier(var);
```

ループの依存解析：基礎

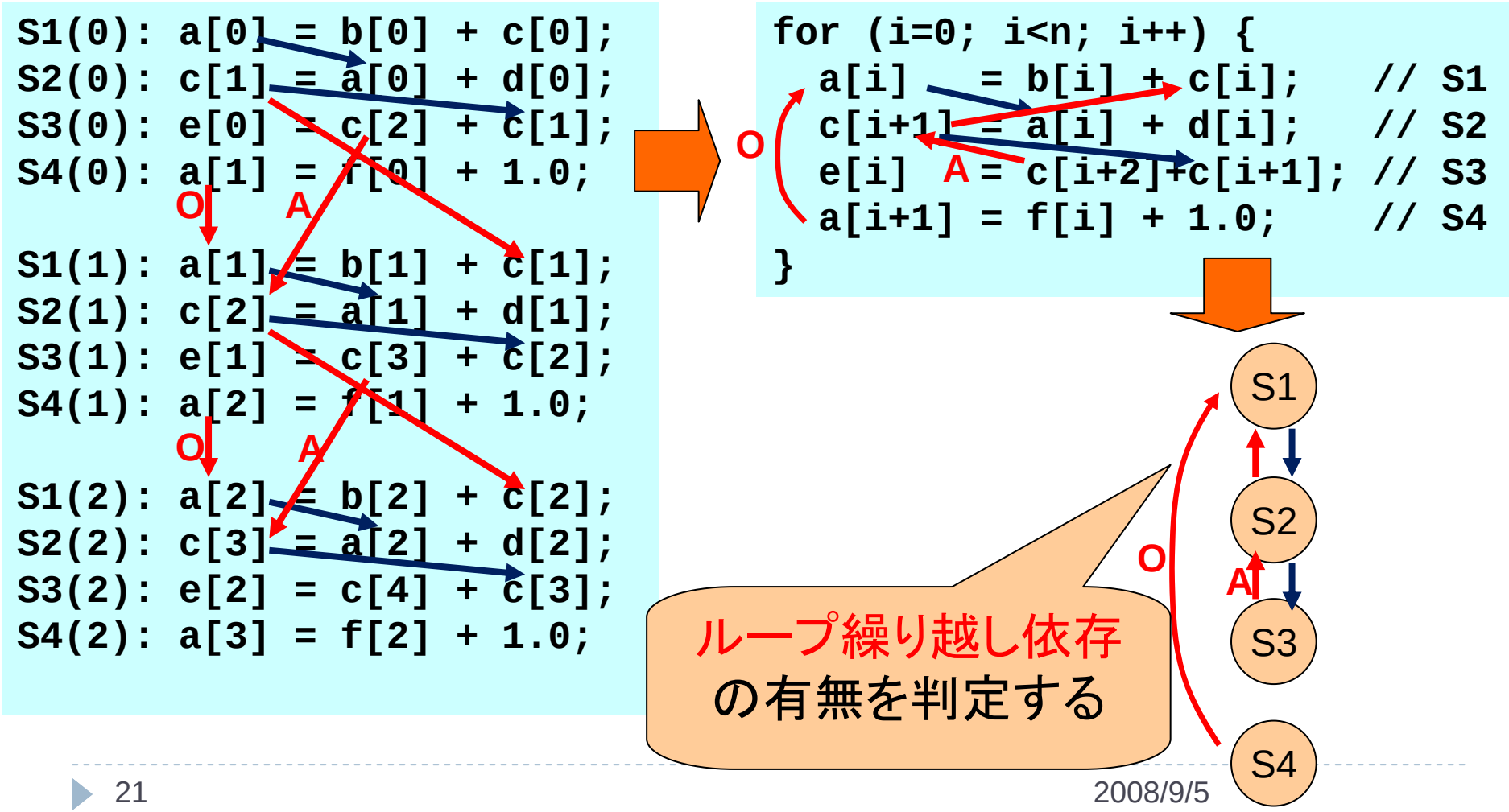
```
for (i=0; i<n; i++) {
  a[i]   = b[i] + c[i];   // S1
  c[i+1] = a[i] + d[i];   // S2
  e[i]   = c[i+2]+c[i+1]; // S3
  a[i+1] = f[i] + 1.0;    // S4
}
```



```
S1(0): a[0] = b[0] + c[0];
S2(0): c[1] = a[0] + d[0];
S3(0): e[0] = c[2] + c[1];
S4(0): a[1] = f[0] + 1.0;
          O    A
S1(1): a[1] = b[1] + c[1];
S2(1): c[2] = a[1] + d[1];
S3(1): e[1] = c[3] + c[2];
S4(1): a[2] = f[1] + 1.0;
          O    A
S1(2): a[2] = b[2] + c[2];
S2(2): c[3] = a[2] + d[2];
S3(2): e[2] = c[4] + c[3];
S4(2): a[3] = f[2] + 1.0;
```

ループ繰り越し依存
(loop carried dependence)

ループの依存解析：依存グラフ

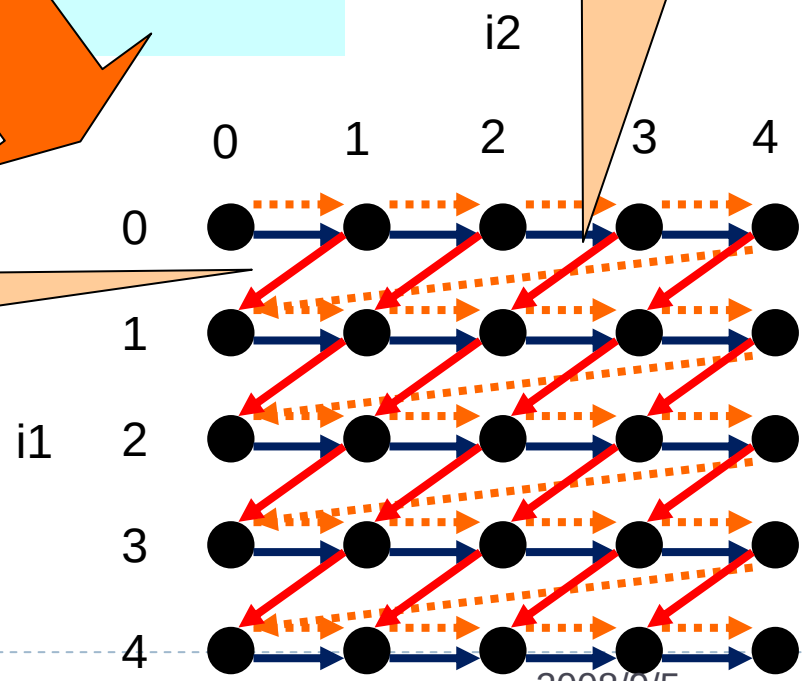


ループの依存解析：イタレーション空間

```
for (i1=0; i1<5; i1++) {
  for (i2=0; i2<5; i2++) {
    a[i1+1,i2] = b[i1,i2] + c[i1,i2];
    b[i1,i2+1] = a[i1,i2+1] + 1;
  }
}
```

b[]はi2のループに関して依存

a[]はi1のループに関して依存



ループの依存解析：gcdテスト

```
for (i=p; i<=q; i++) {// p,q定数
  a[c*i+c0] = ...      // S1
  ... = a[d*i+d0]...   // S2
}
```

a[c*i1+c0] と a[d*i2+d0]は同じアドレスを指すかどうか?

c*i1-d*i2=d0-c0
(c*i1+c0=d*i2+d0)は
p<=i1,i2<=qの範囲で
整数解を持つかどうか?

gをc,dの最大公約数(gcd)と
したとき、d0-c0がgで割り切れれば
整数解を持つ

n重ループの場合の一般化

a1*x1+a2*x2+...+an*xn=b（ディオファンタス方程式）の
整数解の有無を判定

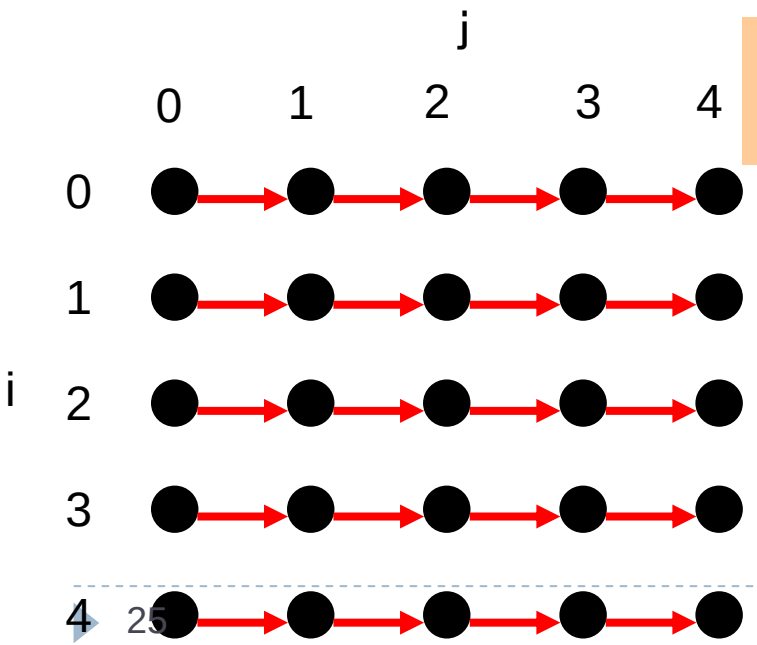
実際に解を求めると依存の種類と繰り返しの位置もわかる

その他ループ（配列）の依存解析

- ループ（配列）の依存解析に関しては以下が詳しい
  - Michael Wolfe, “High Performance Compilers for Parallel Computing”, The Addison-Wesley Publishing Company
  - Intel CompilerではFourier-Motzkin Projectionを使っているらしい
- 基本的には線形方程式（あるいは不等式）の整数解の求解問題
- 配列要素の間接アクセス（a[b[i]]のような形式）には無力

ループ変換の例：ループ交換 (loop interchange)

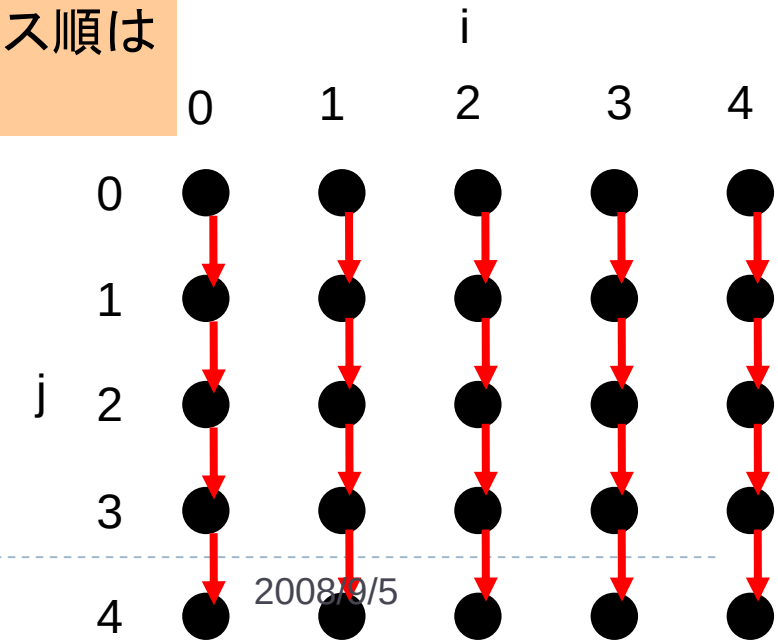
```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    b[i]=b[i]+a[j][i];
  }
}
```



配列bのアクセス順は保存される

```
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    b[i]=b[i]+a[j][i];
  }
}
```

配列aが連続アクセスされる

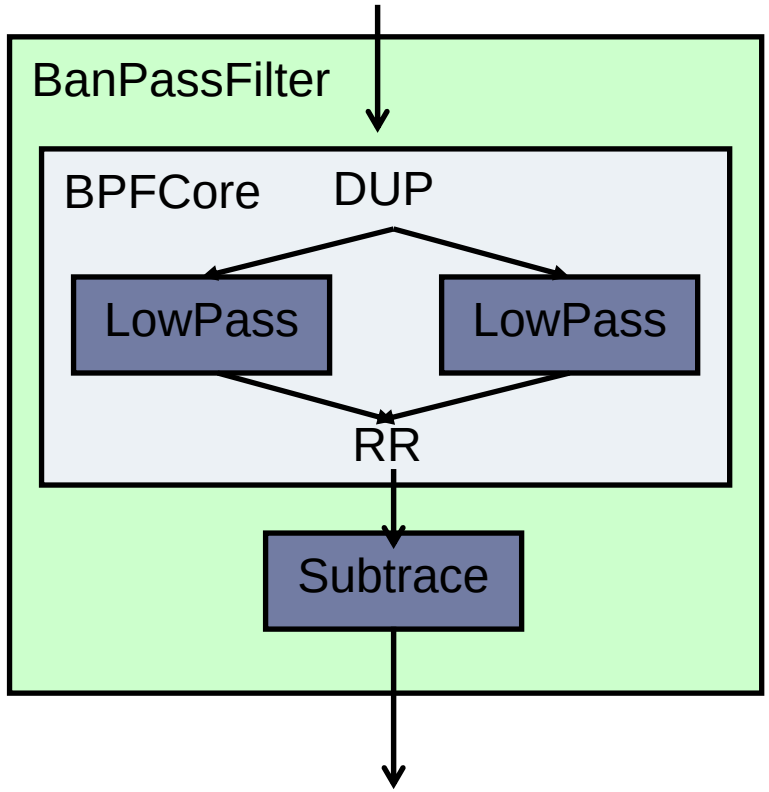


StreamIT

- MITによるdomain specific language
  - ストリーム処理用
  - 文法的にはJavaをベースとしている
- プログラムの構成要素
  - filter
  - pipeline, splitjoin, feedback loop
    - 構成要素の接続方法
- 構成要素間の通信と並列性を明示する

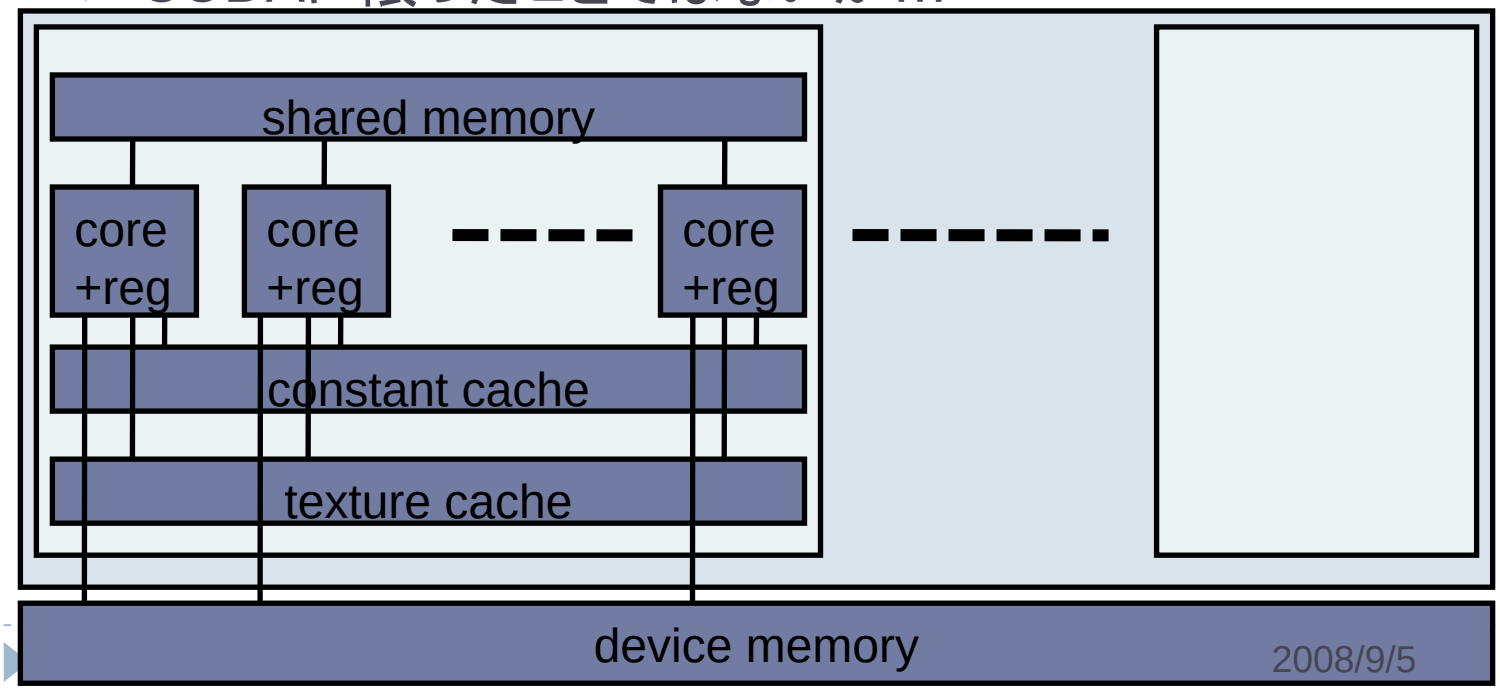
StreamIt Cookbookのサンプルより

```
float->float pipeline BandPassFilter
  (float rate , float low, float high , int taps) {
    add BPFCore(rate, low, high, taps);
    add Subtracter();
}
float->float splitjoin BPFCore
  (float rate , float low, float high , int taps) {
    split duplicate;
    add LowPass(rate, low, taps, 0);
    add LowPass(rate, high, taps, 0);
    join roundrobin;
}
float->float filter Subtracter {
  work pop 2 push 1 {
    push(peek(1) - peek(0));
    pop(); pop();
  }
}
```



Cuda

- NVDIAによるGPU用プログラミングモデルと
  プログラミング環境
- Cに対する簡易な拡張言語
- SIMD型（ドキュメントではSIMT型）のプログラミングモデル
- 性能向上にはメモリ階層を意識する必要あり
  - CUDAに限ったことではないが…



X10, Habanero

- X10
  - IBMによるJavaをベースとした並列プログラミング言語
- Habanero
  - 米RICE大学によるX10の後継プログラム言語
- 特徴
  - 軽量なスレッド生成によるタスクベースの並列処理とループ並列処理
  - 共有メモリモデルと分散メモリモデルを混在したプログラミングモデル

X10, Habaneroのプログラムの例

```
void refine(final int n, final int l, final int
nmax){
  left=new Tree(this, 2.0*i);
  right=new Tree(this, 2.0*i+1);
  final nullable Tree ll=left, rr=right;
  if (n < (nmax-1)){
    async{ll.refine(n+1,2*l,nmax);}
    async{rr.refine(n+1,2*l,nmax);}
  }
  if (n < nmax) data = null;
…
}
```

再帰的なタスク並列処理

```
int iteres=0; delta=epsilon+1;
while (delta>epsilon) {
 finish {
  foreach (point[j]:[1:n]) {
   newA[j] = (oldA[j-1]+oldA[j+1])/2.0f;
   diff[j] = Math.abs(newA[j]-oldA[j]);
   }
  }
  delta = diff.sum(); iters++;
  temp = newA; newA = oldA; oldA =
temp;
}
```

foreachによる
ループ並列処理

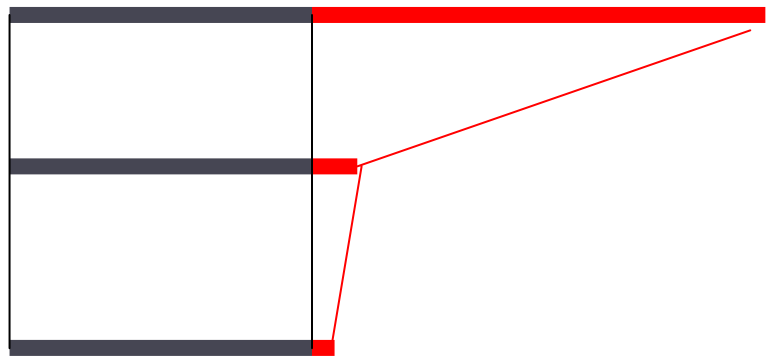
並列処理のポイント

- チューニングの労力と得られる効果のトレードオフ
- 処理の粒度
- 負荷分散
- メモリの配置場所
- 同期・通信

アムダールの法則

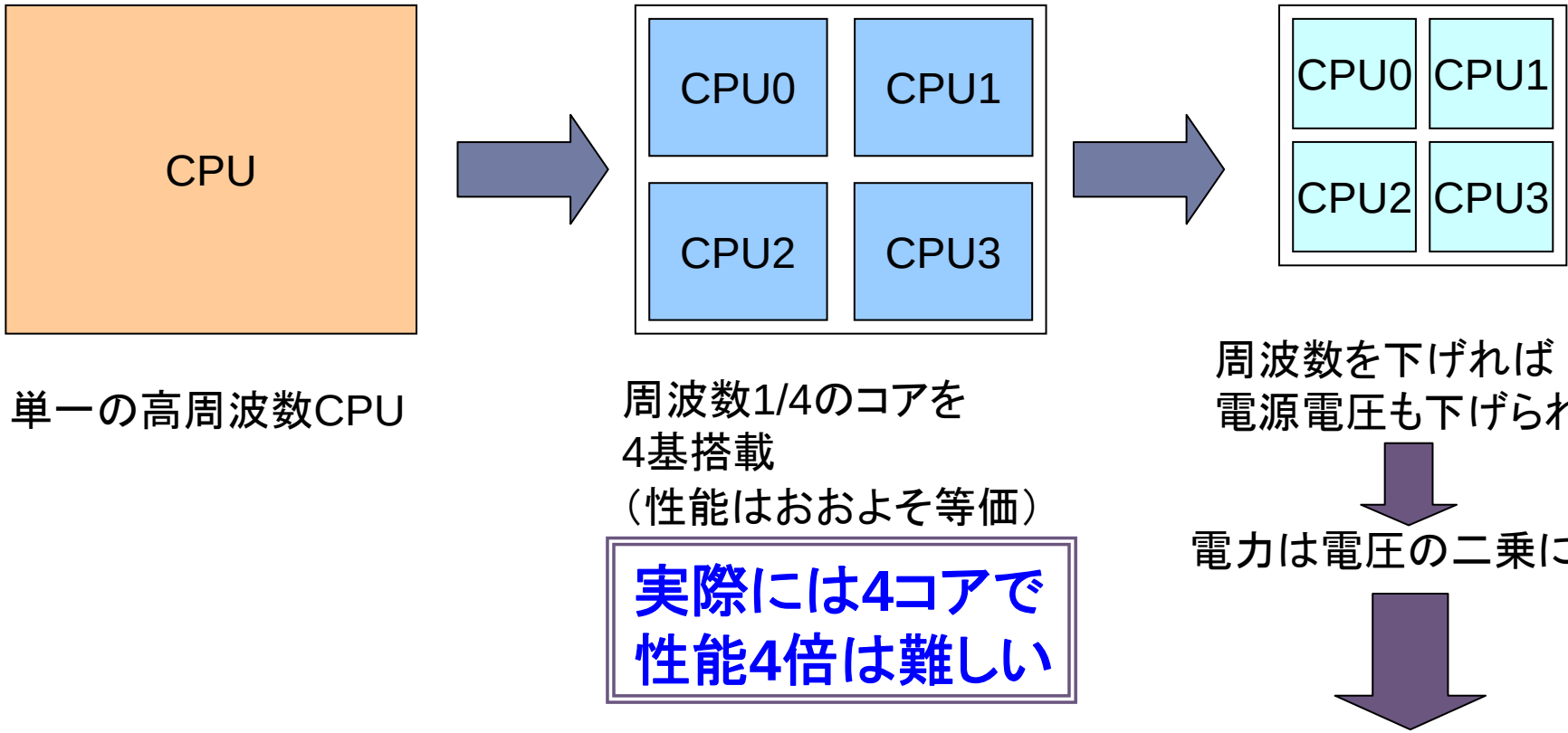
ex.
最適化可能な箇所の割合:
　　総実行時間の60%
最適化による速度向上率: 10倍
　-> 1/(0.4+0.6x(1/10)) = 2.174

最適化による20倍になったとしたら？
　-> 1/(0.4+0.6x(1/20)) = 2.326



並列化と低消費電力化

- ダイナミック電力: $P = \alpha C V^2 f$



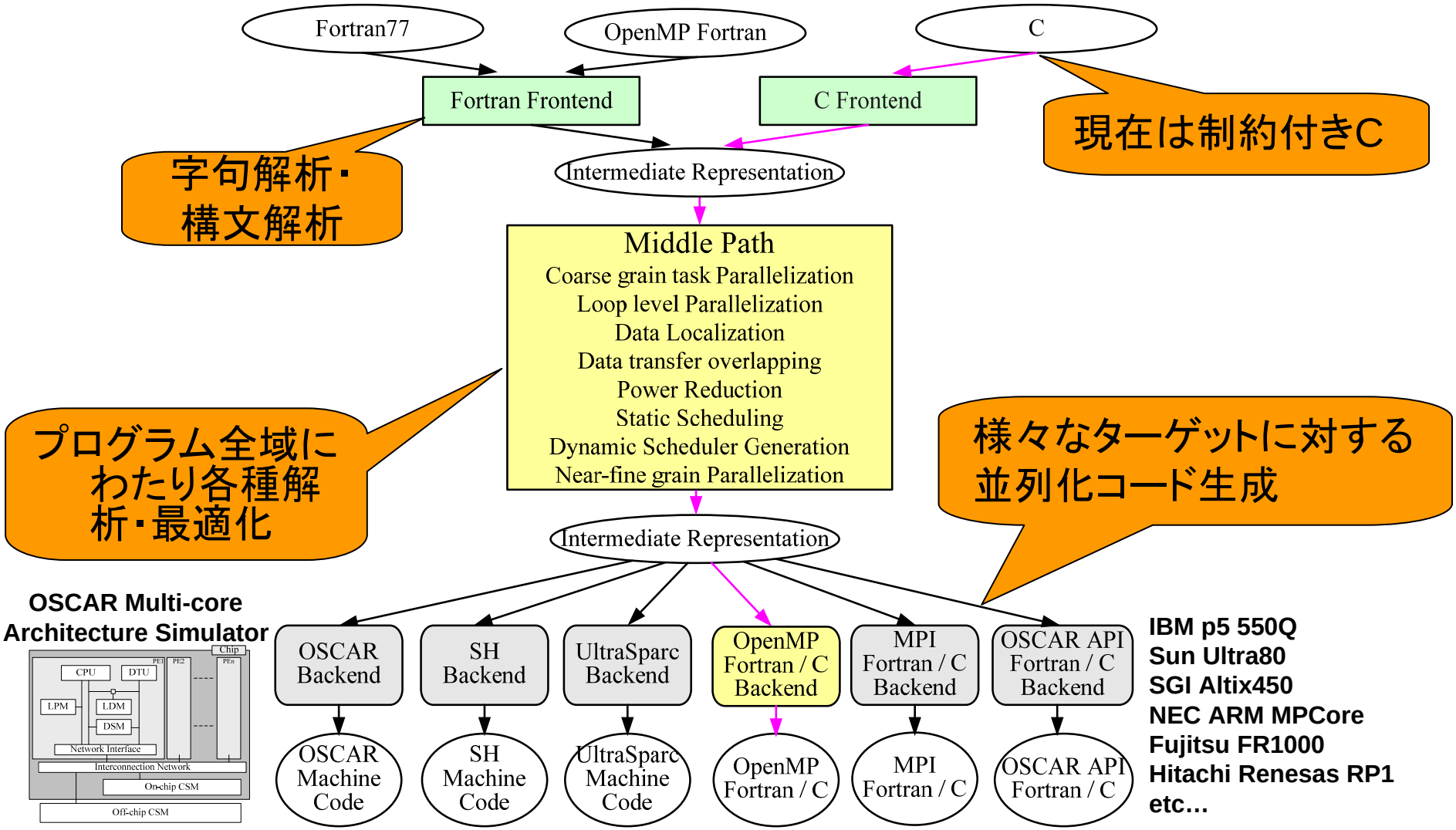
演習

- 並列化コンパイラで並列化しやすいアプリケーションは？またその理由は？
  - 画像フィルタ、画像認識
  - 動画像コーデック
  - ワープロ
  - データ検索

OSCAR自動並列化コンパイラ

- 早稲田大学 笠原・木村研で開発している自動並列化コンパイラ
- マルチグレイン並列化
  - プログラム全域にわたる並列化
- メモリ最適化
  - キャッシュ・ローカルメモリ利用の最適化
- データ転送最適化
  - データ転送隠蔽技術
- 低消費電力化
- ヘテロジニアス・スケジューリング

OSCAR自動並列化コンパイラの構成

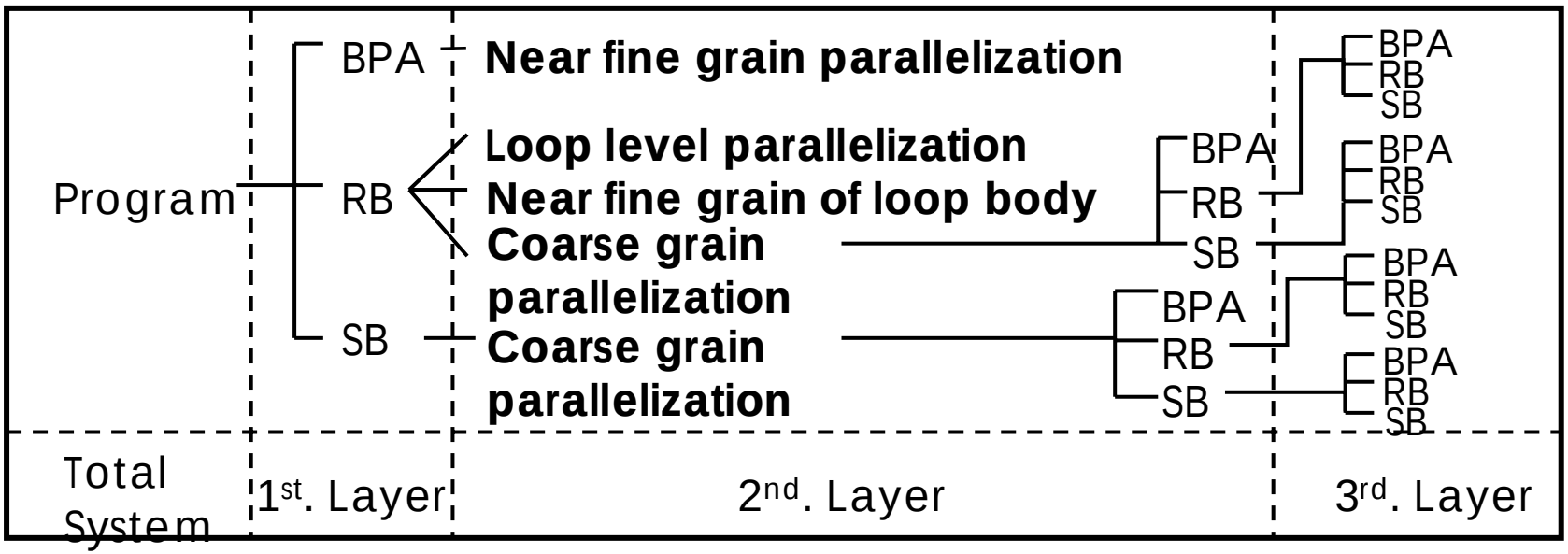


マルチグレイン並列処理

- プログラム全域にわたる最適化
  - ループイタレーションレベル並列化
    - 通常の並列化コンパイラ
  - 粗粒度タスク並列化
  - 近細粒度並列化

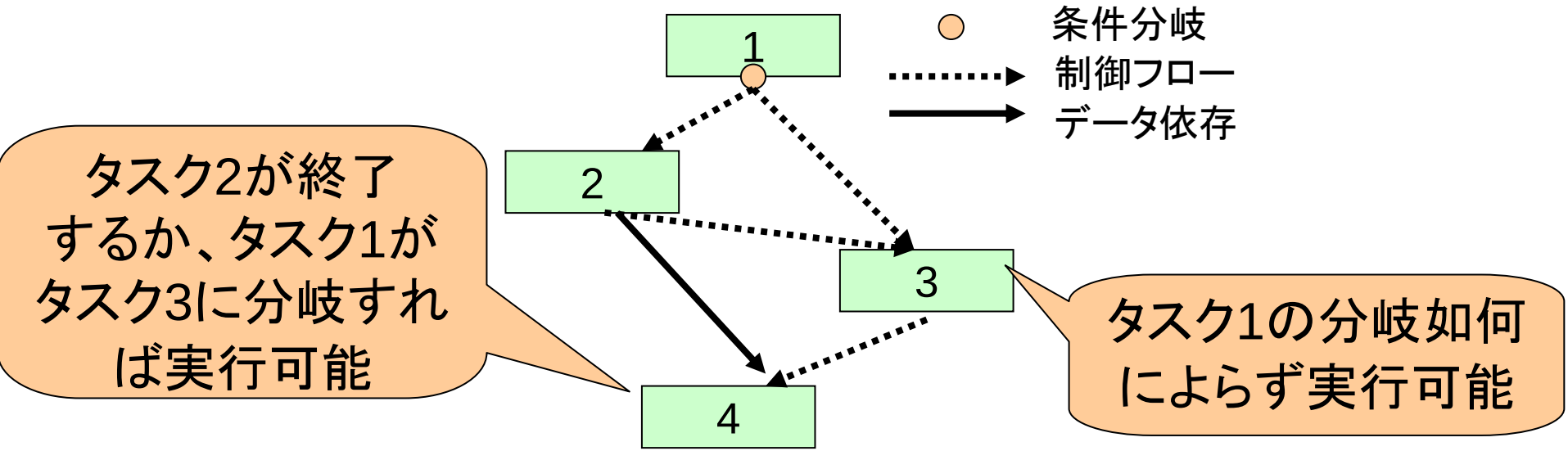
粗粒度タスク並列処理

- **マクロタスク**
  - 擬似代入文ブロック(BPA)、基本ブロック(BB)
  - 繰り返しブロック(RB)
  - サブルーチンブロック(SB)

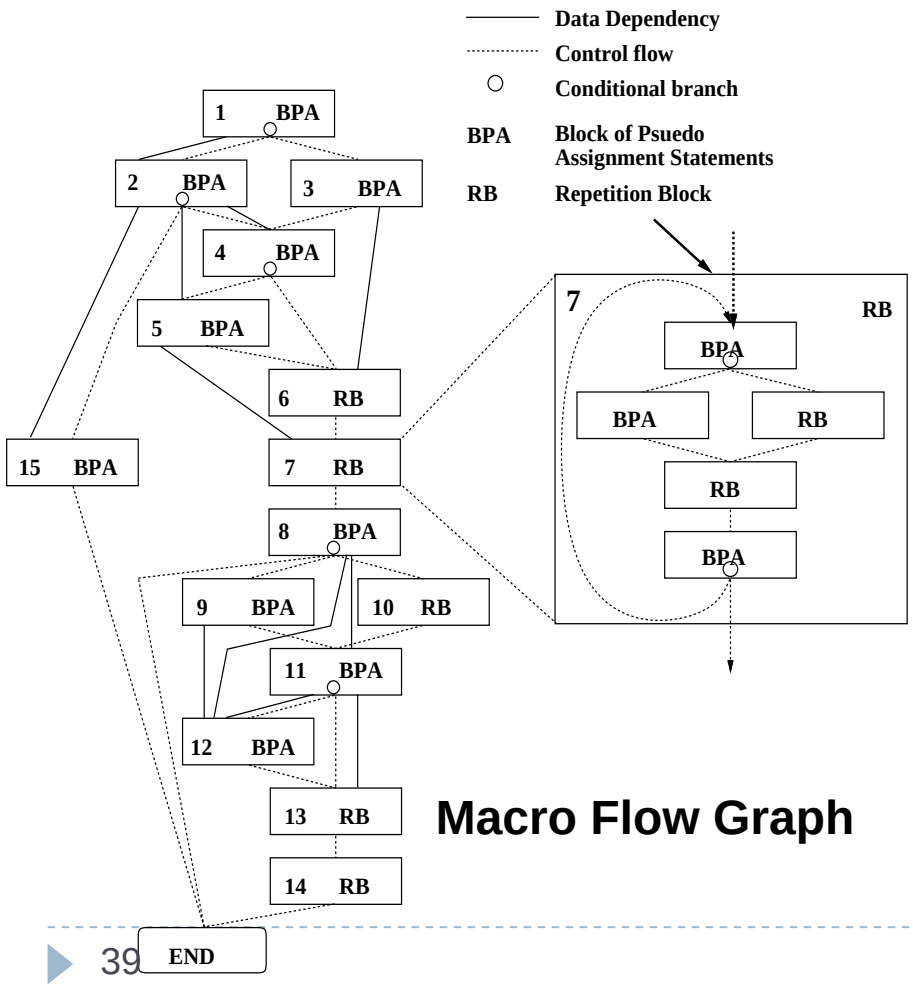


最早実行可能条件解析

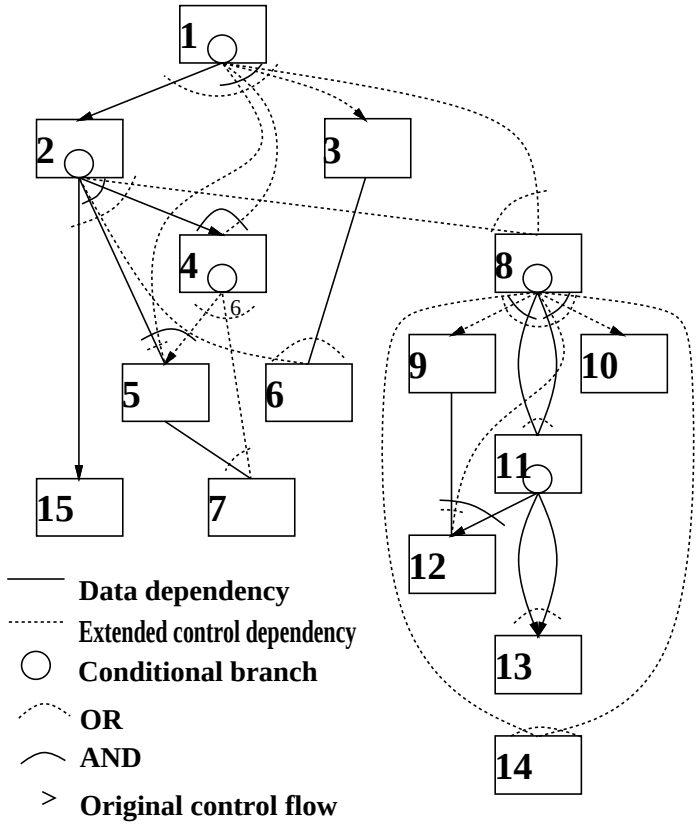
- 粗粒度タスクの実行条件
  - and
    - 着目タスクの実行が確定
    - 着目タスクのデータがそろう
      - データ依存タスクの実行が終了
      - or
        - データ依存タスクが実行されないことが確定



マクロフローグラフと
マクロタスクグラフ



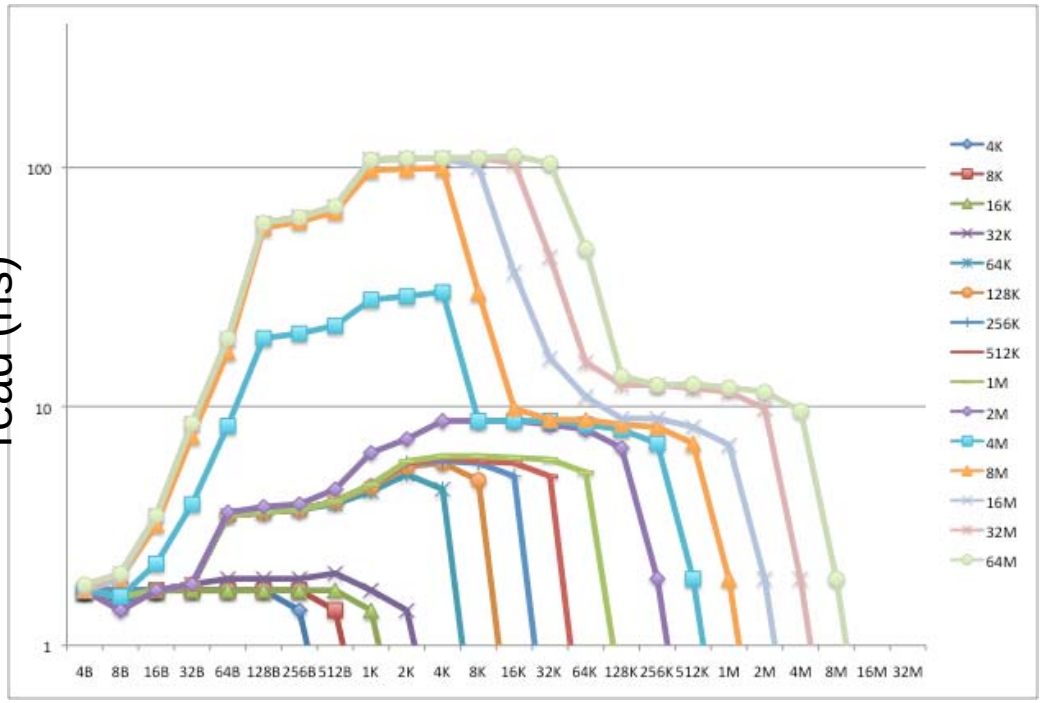
Macro Flow Graph



Macro Task Graph

メモリの効率的な利用

- メモリはどんどん遠くなる
  - 近接メモリ（キャッシュ、スクラッチパッドメモリ等）の効率的な利用が重要
- チップ内外を問わずバンド幅は限られる
  - コア間共有データ、コア固有データをどのように保持するべきか？
  - データ転送はどうする？

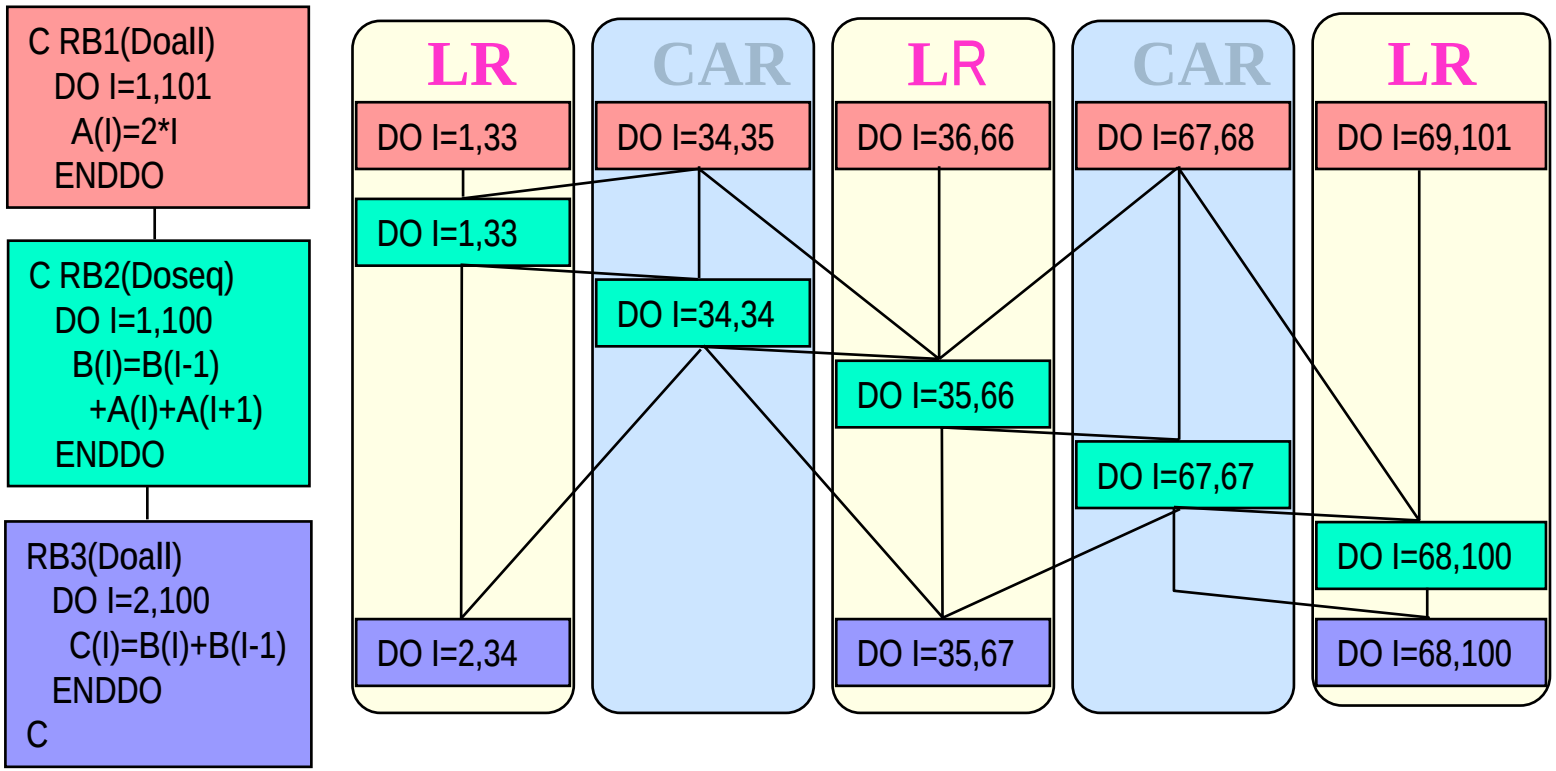


ストライドを変えたメモリアクセスによりキャッシュ周りのパラメータを測定した結果（参考：ヘネパタ第4版）

測定環境：
Intel xeon x5365@3GHz (4x2 core)
L1Dcache 32KB
L2cache 4MB/2core

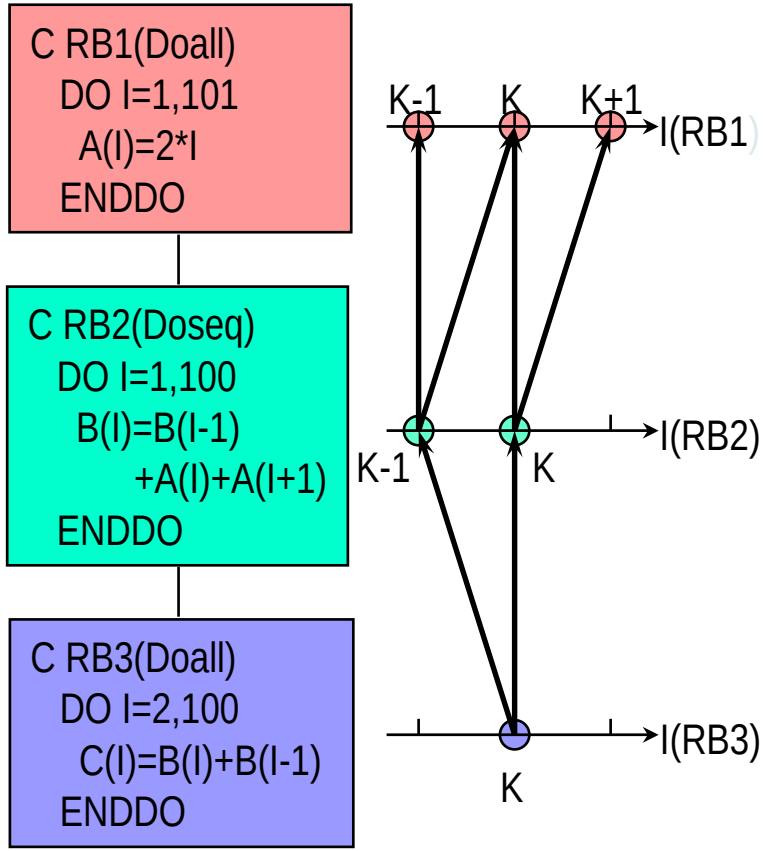
データローカライゼーション: 整合分割

- 複数のループをループ間のデータ依存を考慮しつつCARと LRに分割する
  - ローカルメモリあるいはキャッシュを介してLRのデータをループ間で授受する
  - LR: Localizable Region, CAR: Commonly Accessed Region



ループ間データ依存解析

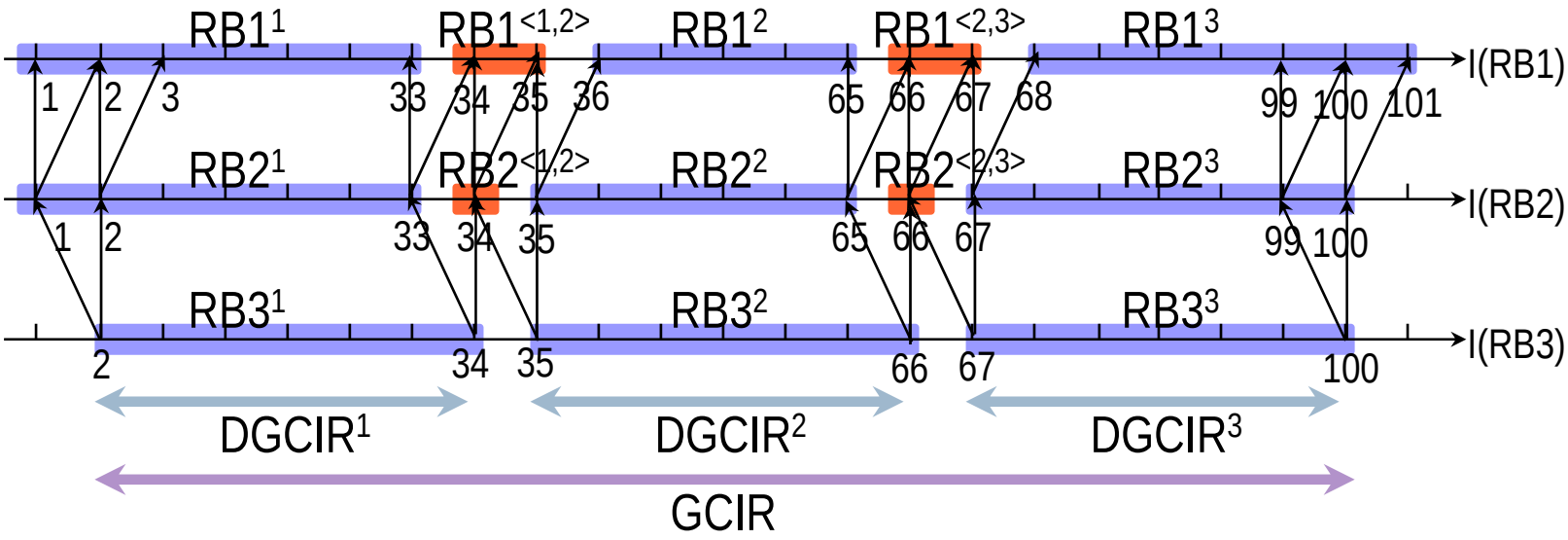
- 右図においてRB3を標準ループとする
- 標準ループの繰り返しが依存する他のループの繰り返しを解析する
  - 例：RB3のK番目のループはRB2のK-1,K番目のループに、RB1のk-1, K, K+1番目のループにそれぞれ依存する



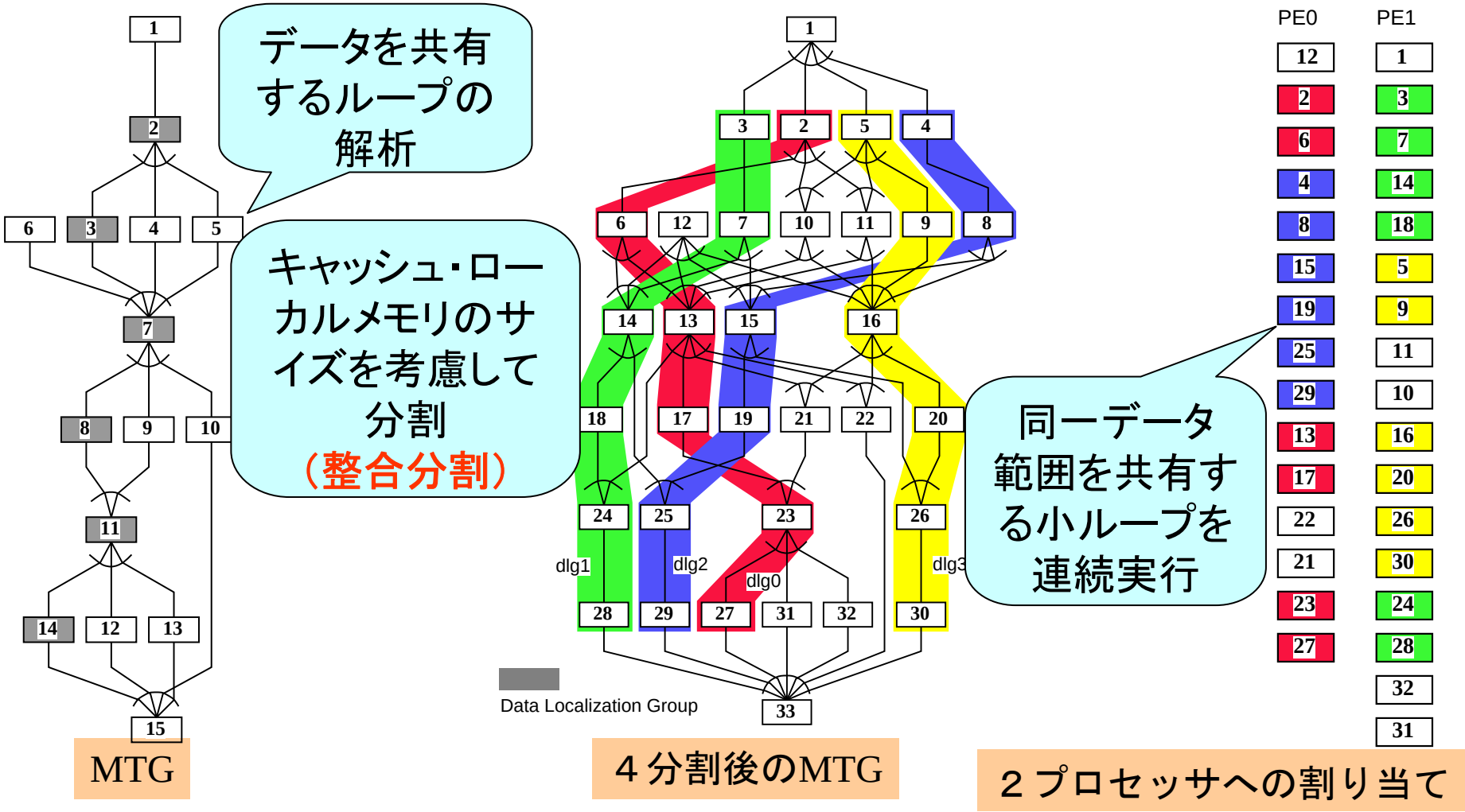
Example of TLG

TLG内ループ分割

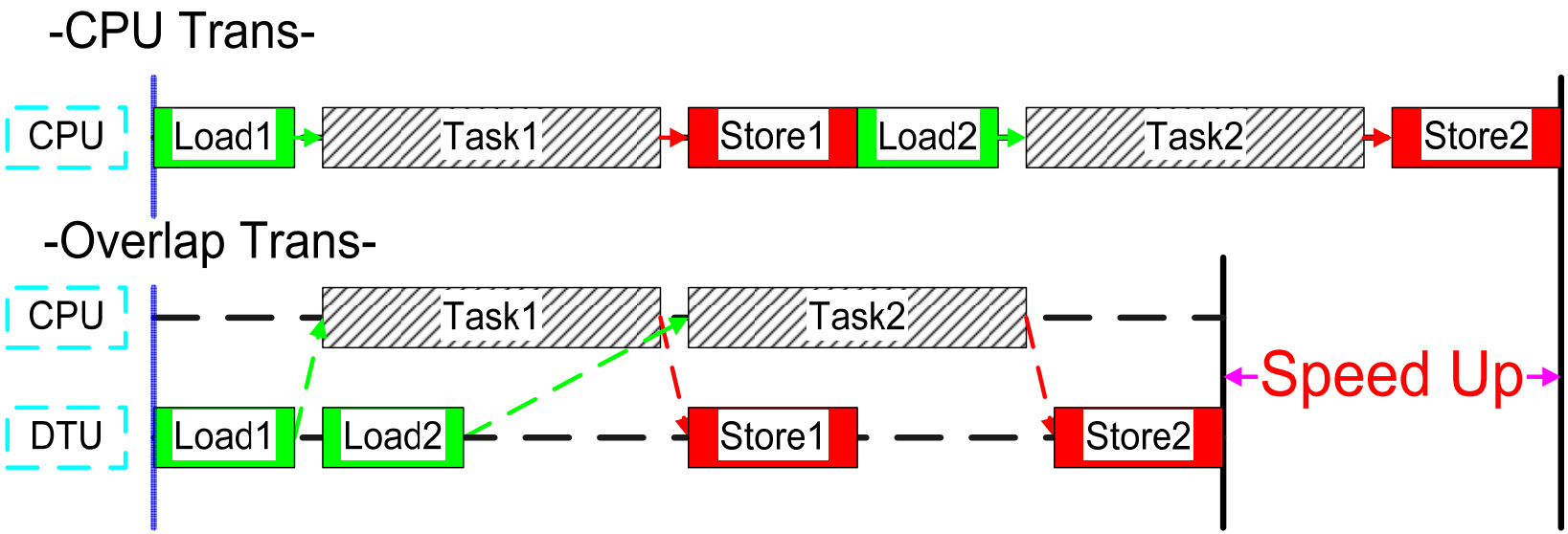
- GCIR（グループ変換インデクス範囲）をDGCIR^{p}(1≦p≦n)（分割グループ変換インデクス範囲）に分割する
  - n: 分割数
- CAR: DGCIR^{p}とDGCIR^{p+1} が依存している領域
- LR: DGCIR^{p} のみが依存している領域



メモリ最適化技術


データを共有するループの解析
キャッシュ・ローカルメモリのサイズを考慮して分割
（整合分割）
同一データ範囲を共有するループを連続実行
PE0
PE1
dlg1
dlg2
dlg0
dlg3
Data Localization Group
MTG
４分割後のMTG
２プロセッサへの割り当て

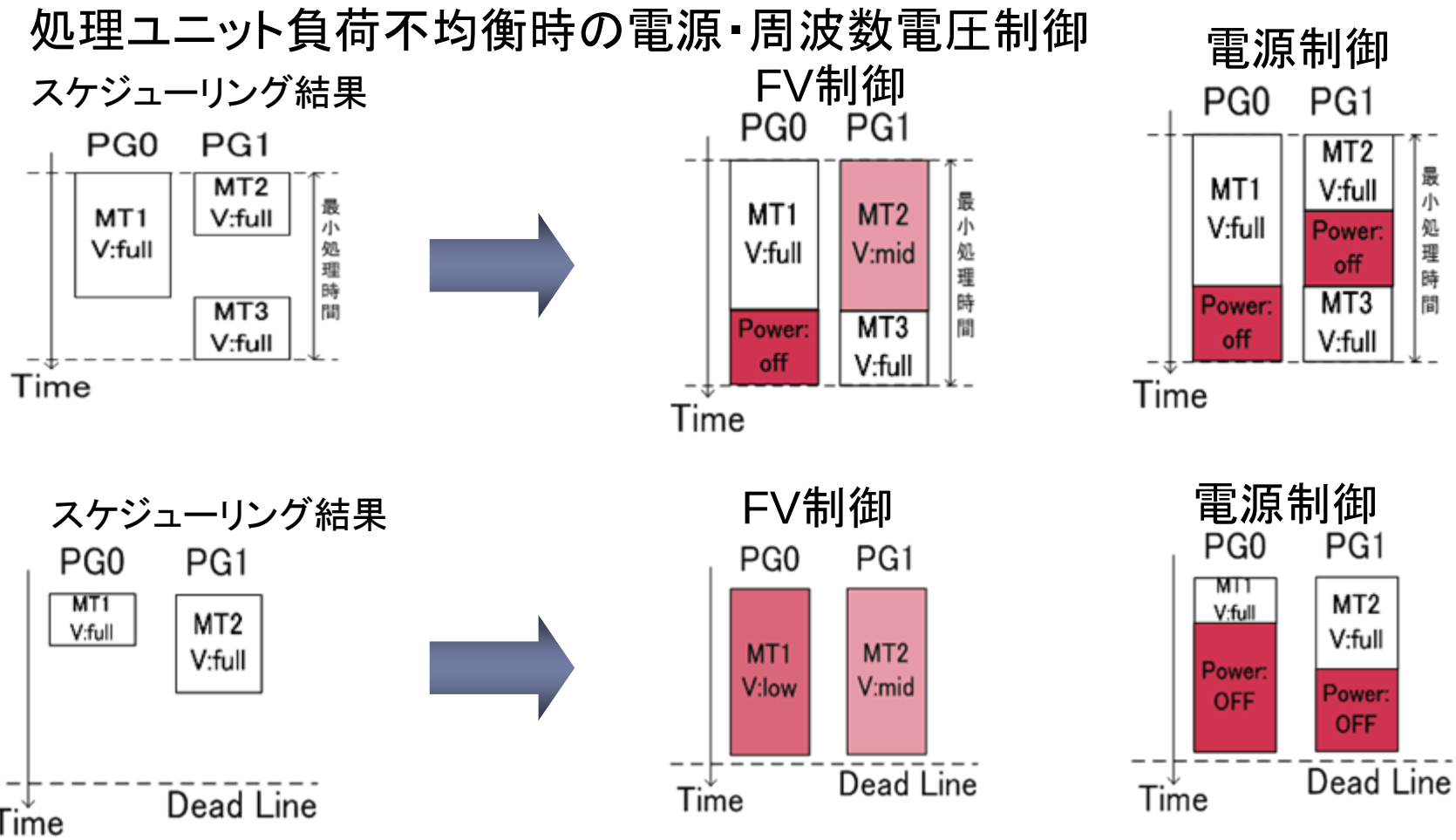
データ転送最適化技術



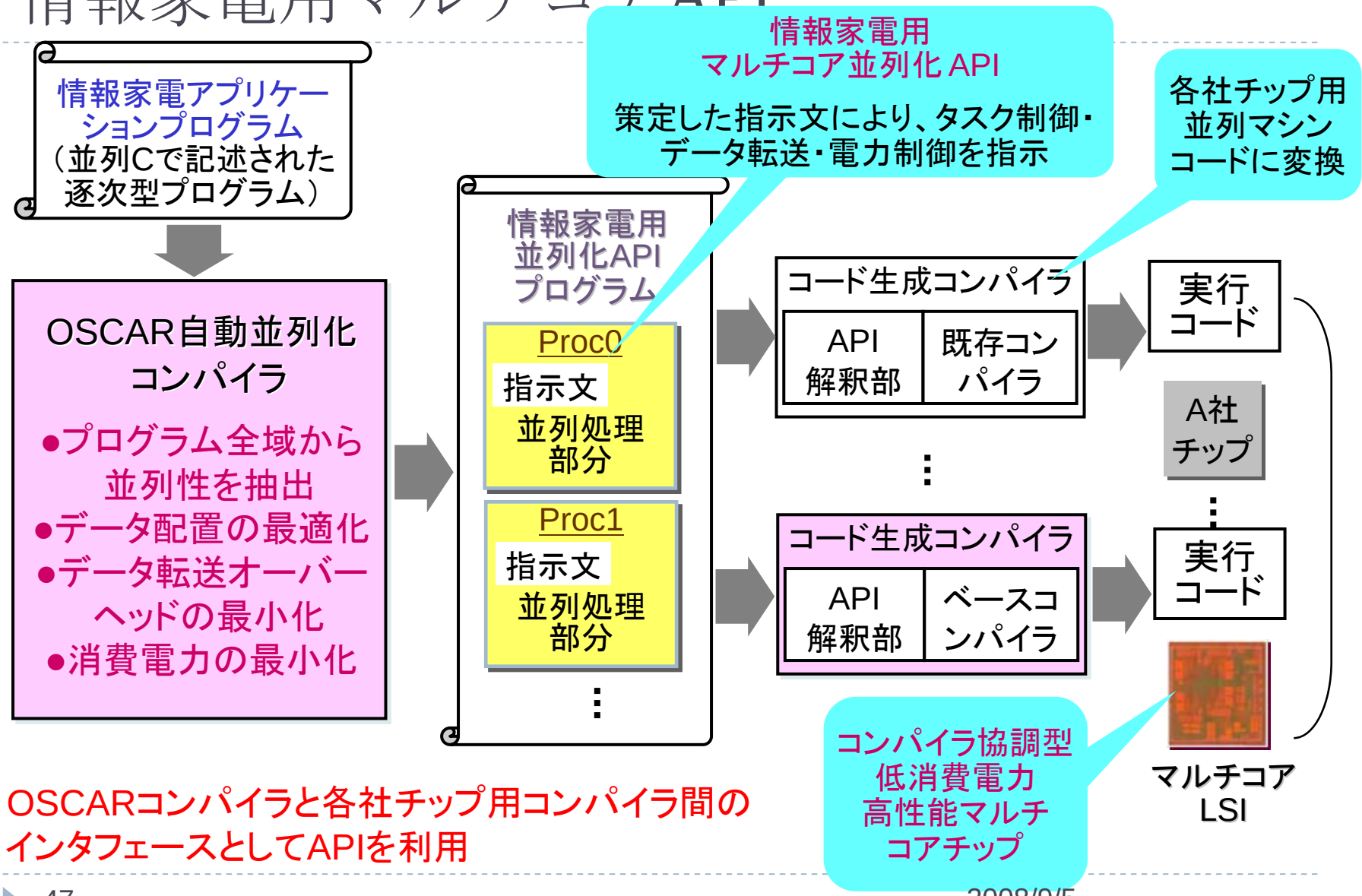
- CPU処理と非同期に動作可能なDTU（データ転送ユニット）を利用してデータ転送をオーバーラップ
- データ転送オーバヘッドを隠蔽し速度向上を得る

低消費電力化技術

- 処理ユニット負荷不均衡時の電源・周波数電圧制御

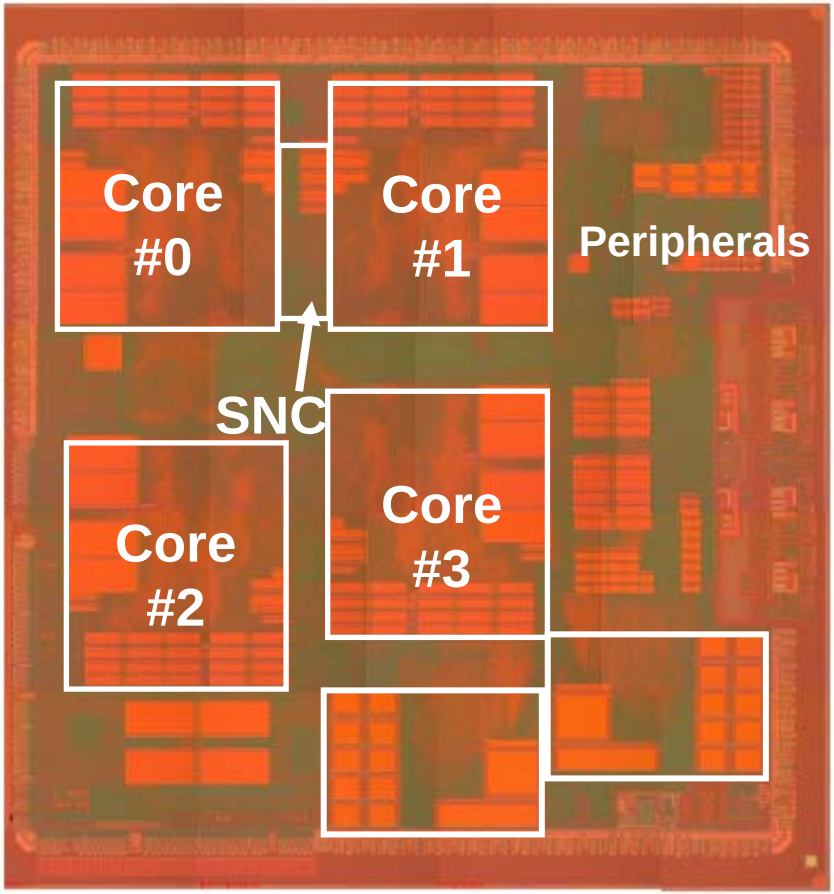


情報家電用マルチコアAPI



OSCARコンパイラと各社チップ用コンパイラ間のインタフェースとしてAPIを利用

情報家電用マルチコアプロセッサ： RP1



SH4A **マルチコア**SoC**チップ写真**

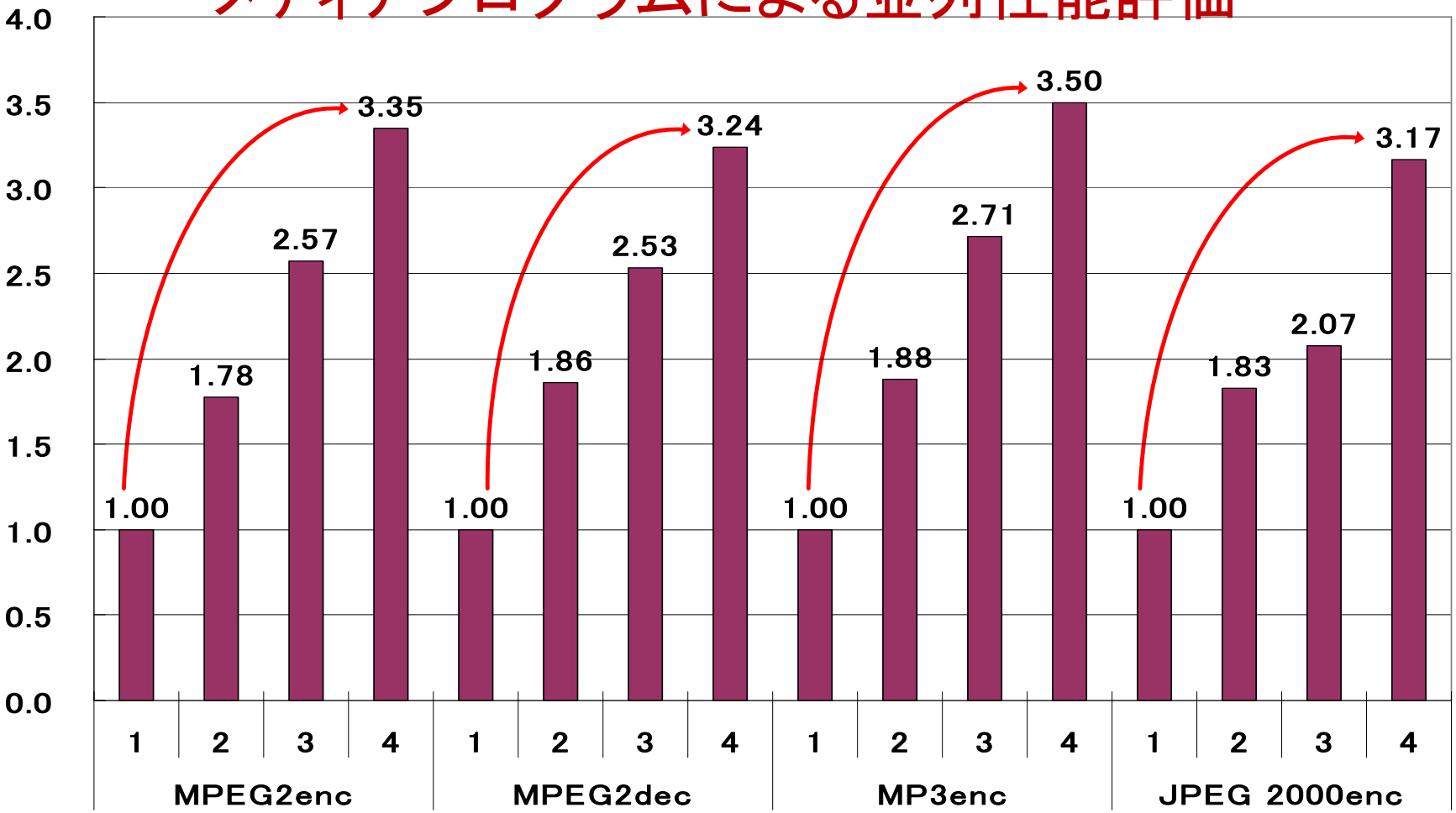
ISSCC07**発表**: ISSCC07 **論文番号**5.3, Y. Yoshida, et al., “A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption”

チップ諸元

Process Technology	90nm, 8-layer, triple-Vth, CMOS
Chip Size	97.6mm^2 (9.88mm x 9.88mm)
Supply Voltage	1.0V (internal), 1.8/3.3V (I/O)
Power Consumption	0.6 mW/MHz/CPU @ 600MHz (90nm G)
Clock Frequency	600MHz
CPU Performance	4320 MIPS (Dhrystone 2.1)
FPU Performance	16.8 GFLOPS
I/D Cache	32KB 4way set-associative (each)
ILRAM/OLRAM	8KB/16KB (each CPU)
URAM	128KB (each CPU)
Package	FCBGA 554pin, 29mm x 29mm

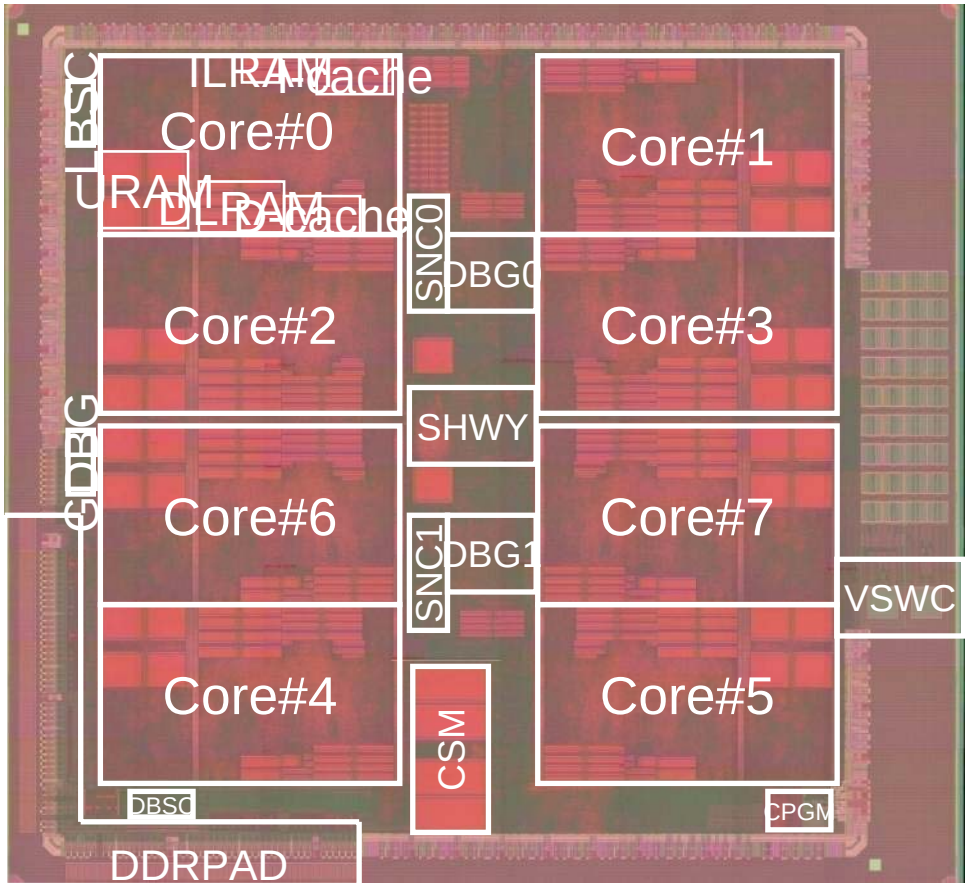
情報家電用マルチコアプロセッサ： RP1

メディアプログラムによる並列性能評価



1プロセッサと比較して、4プロセッサで平均3.31倍の速度向上

情報家電用マルチコアプロセッサ： RP2



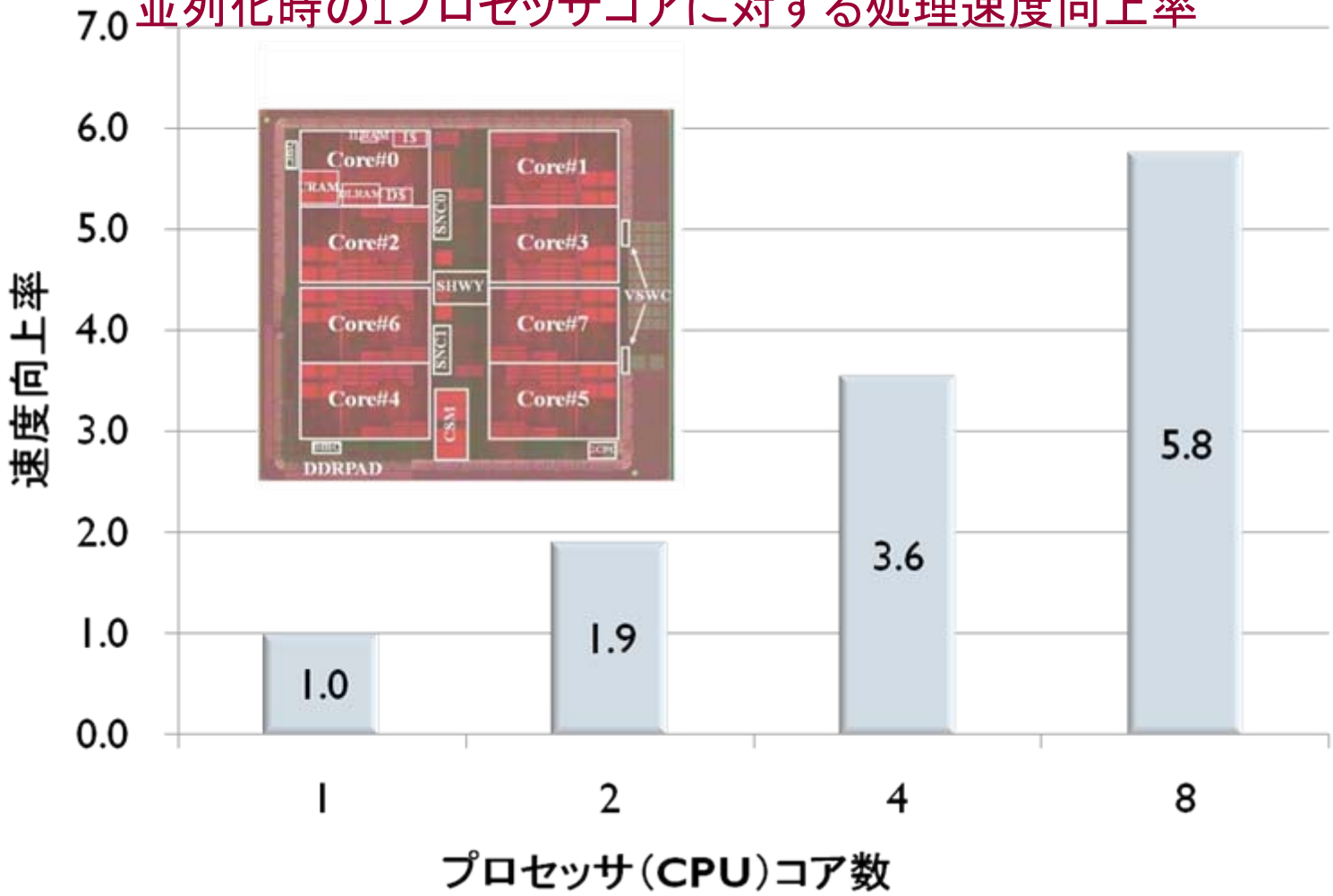
8コア集積マルチコアLSIチップ写真

ﾌﾟﾛｾｽ	90nm CMOS, 8層ﾒﾀﾙ, 3種Vth
ﾁｯﾌﾟｻｲｽﾞ	104.8mm² (10.61mm x 9.88mm)
電源電圧	1.0V-1.4V(ｺｱ), 1.8/3.3V(I/O)
動作周波数	600MHz
CPU性能	8640 MIPS (Dhrystone 2.1)
FPU性能	33.6 GFLOPS
低電力制御	・CPU毎に独立した周波数変更 ・CPUコアのクロックを停止するスリープモード ・CPUコアの一部のクロックを停止するがキャッシュコヒーレンシ維持可能なライトスリープモード ・CPUコアの電源供給を停止するフル電源遮断モード ・URAM以外のCPUコアの電源供給を停止するレジューム電源遮断モード

ISSCC08発表: ISSCC08 論文番号4.5, M.ITO, et al., “An 8640 MIPS SoC with Independent Power-off Control of 8 CPUs and 8 RAMs by an Automatic Parallelizing Compiler”

情報家電用マルチコアプロセッサ： RP2

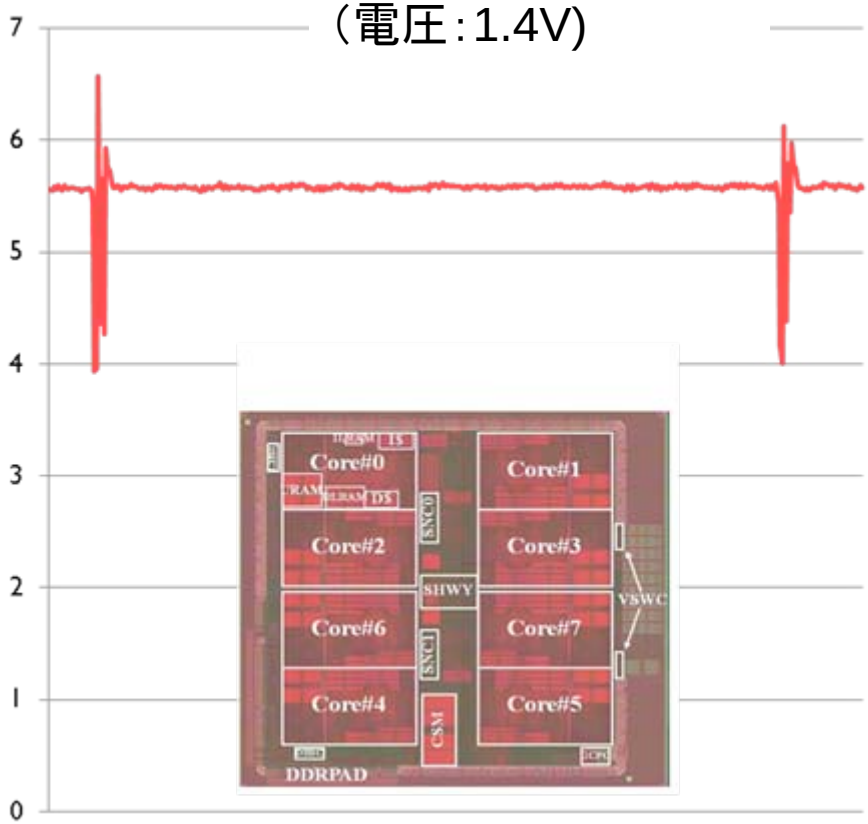
オーディオ圧縮（AAC*エンコード）処理（マルチメディア処理）
並列化時の1プロセッサコアに対する処理速度向上率



情報家電用マルチコアプロセッサ： RP2

セキュアオーディオ圧縮（AACエンコード+AES暗号化）処理を8コアで実行時の消費電力

電力制御なし
（電圧：1.4V)



周波数/電圧・電源制御あり
（電圧：1.0V～1.4V、
レジューム電源遮断モード使用)

平均電力
5.68 [W]

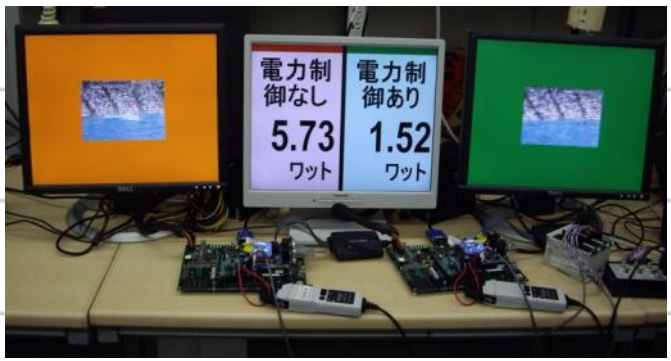
88.3%の電力削減

平均電力
0.67 [W]

情報家電用マルチコアプロセッサ： RP2

動画表示（MPEG2デコード処理）を8コアで実行時の消費電力

電力制御なし
（電圧：1.4V)



周波数/電圧・電源制御あり
（電圧：1.0V～1.4V、
レジューム電源遮断モード使用)

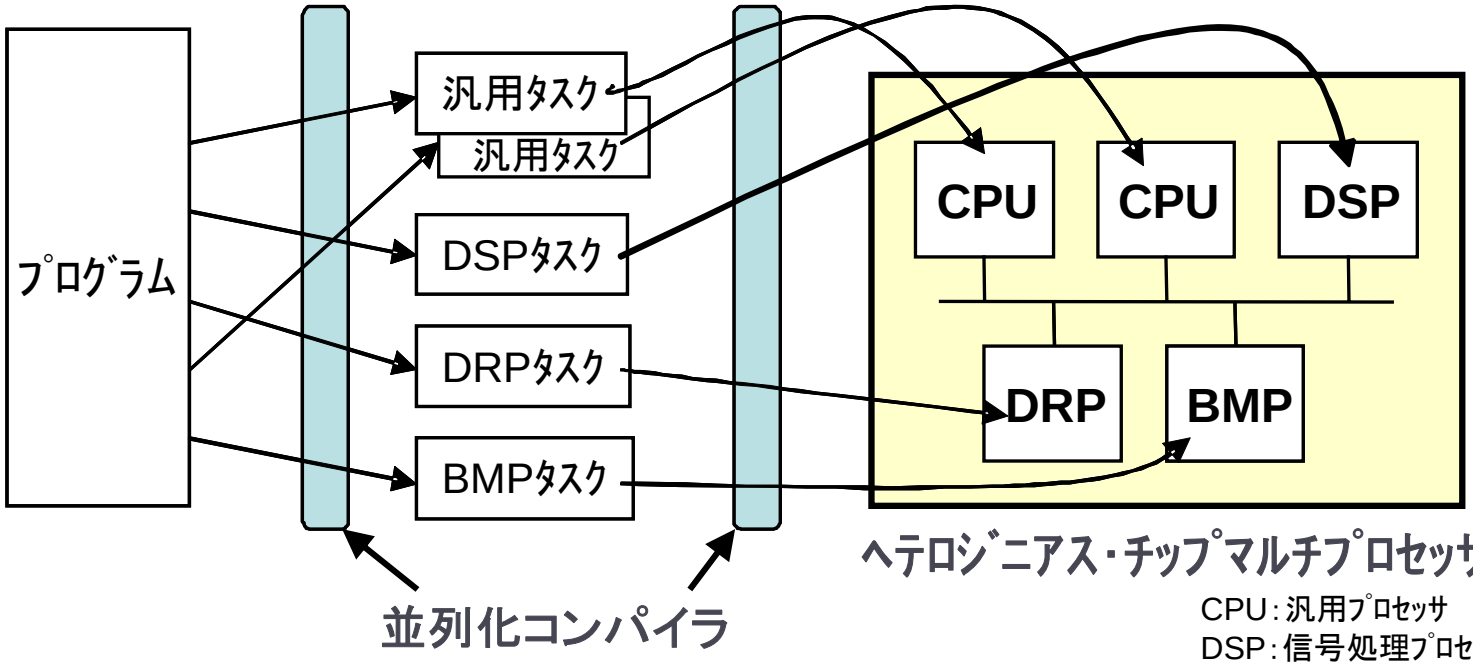
平均電力
5.73 [W]

73.5%の電力削減

平均電力
1.52 [W]

ヘテロジニアスマルチコア

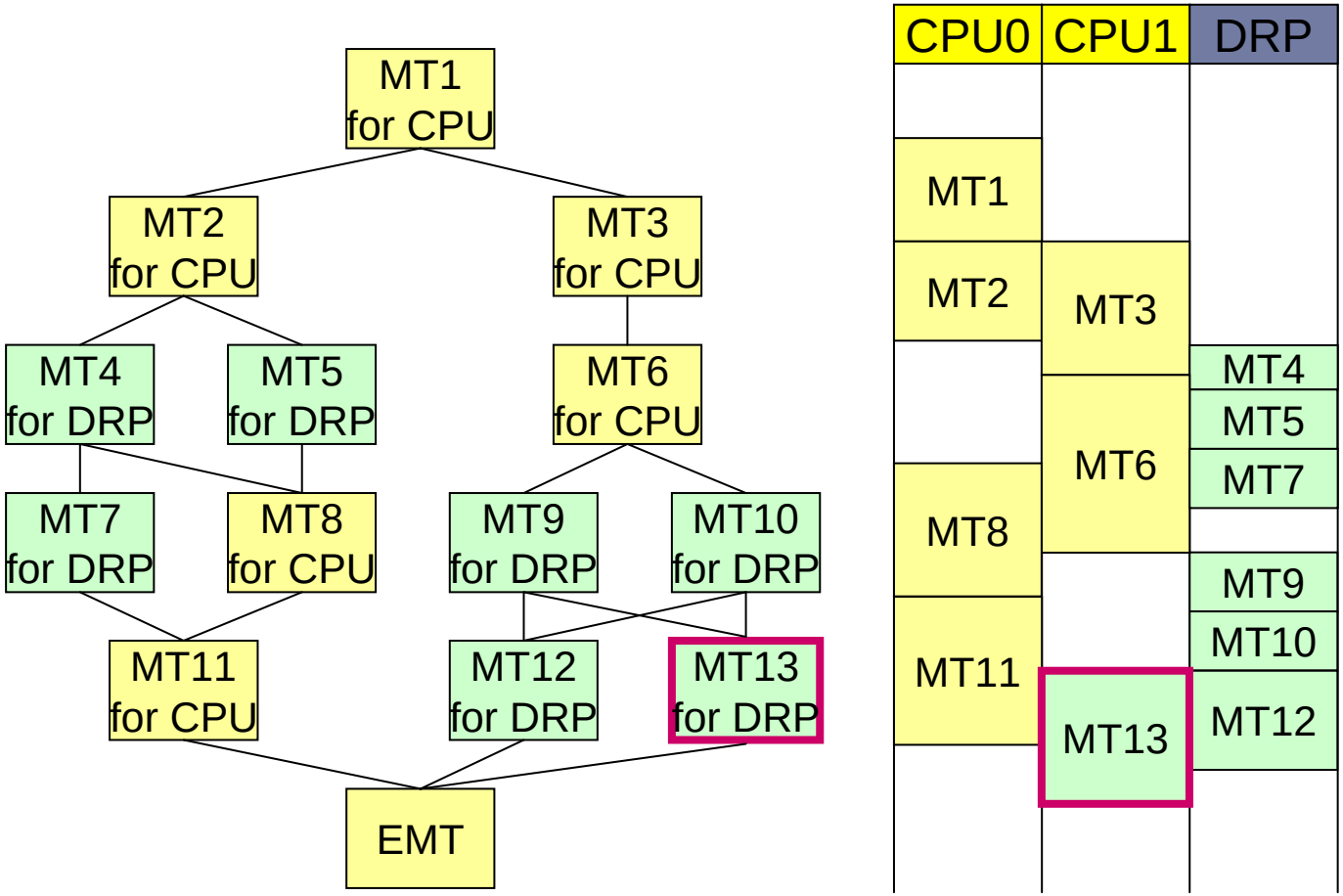
- 多種類の計算エンジン(プロセッサ)を1チップに集積したSoCアーキテクチャ
- プログラムの並列性を抽出し、各プロセッサの特徴に適したタスクの分割と配置を行う並列化コンパイラ



CPU:汎用ﾌﾟﾛｾｯｻ
DSP:信号処理ﾌﾟﾛｾｯｻ
DRP:動的再構成可能ﾌﾟﾛｾｯｻ
BMP:ﾋﾞｯﾄ処理ﾌﾟﾛｾｯｻ

ヘテロジニアスマルチコア

スケジューリングのイメージ



ヘテロジニアスマルチコア

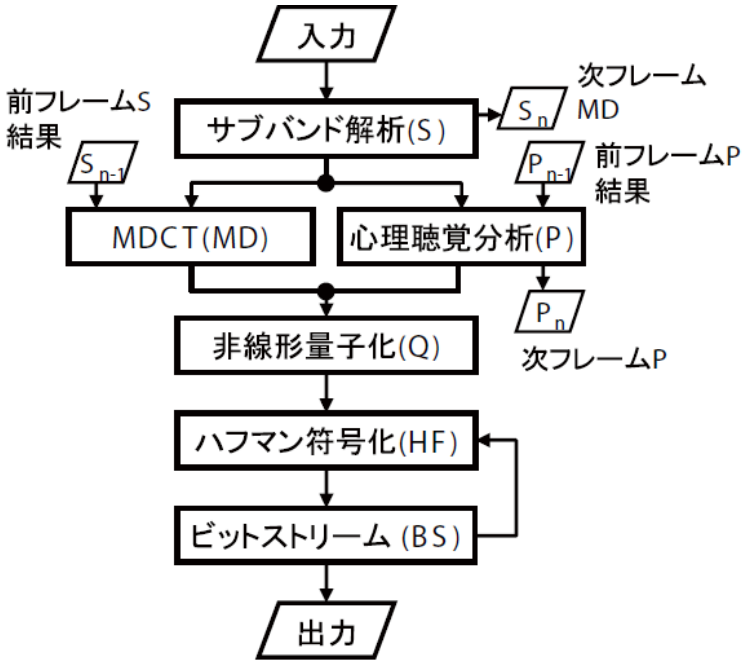
評価条件

- アーキテクチャ
  - 300MHz動作を想定
    - Up to 8Cores
  - 汎用コア：SH4A相当
  - アクセラレータ：DRP（日立FE-GA相当）
  - PE間接続網：3本バス
- メモリレイテンシ
  - LDM : 1 Clock Cycle
  - DSM : 1 Clock Cycle(Local) and 4 Clock Cycles(Remote)
  - CSM : 16 Clock Cycles(Off Chip)
- 評価アプリケーション(MP3エンコーダ)
  - UZURA MPEG1/LayerIII encoder in FORTRAN90 を参照実装
  - アクセラレータタスクは指示文により指定
    - サブバンド解析の一部、MDCT、心理聴覚分析、非線形量子化
  - 入力データ：32フレーム
    - 後半16フレームのサブバンド解析～符号化処理までを評価対象とする

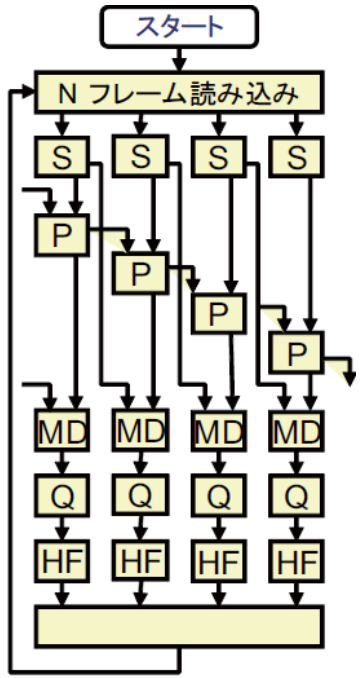
ヘテロジニアスマルチコア

評価アプリケーション

- MP3エンコード処理はフレーム単位
  - フレーム間の並列性を利用することで性能向上
    - 16フレーム分を並列に処理する形式にアンローリング

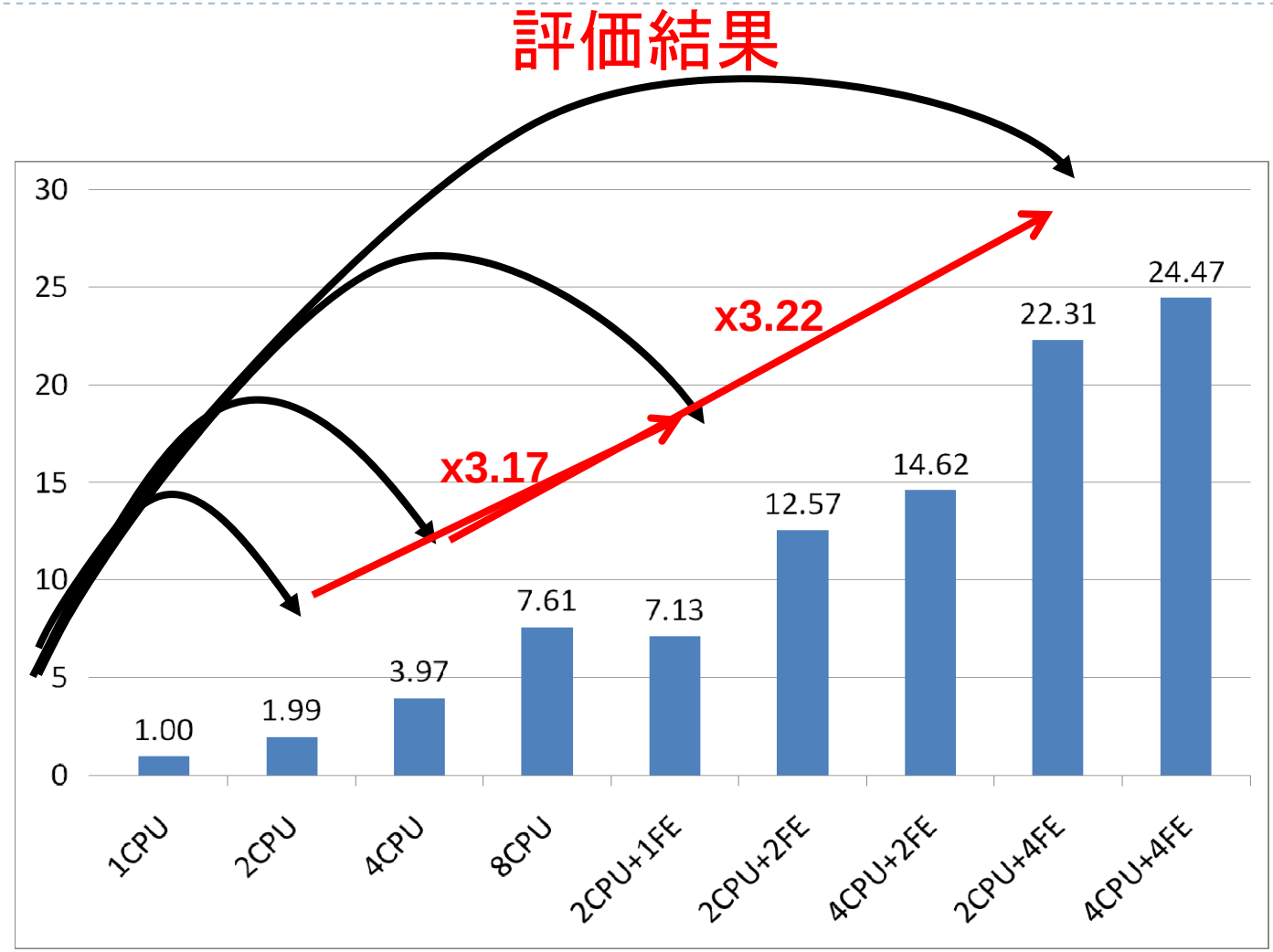


(a)



(b)

ヘテロジニアスマルチコア



演習

- 今後並列化コンパイラで並列化が必要と思われるアプリケーションは？
  - 画像フィルタ、画像認識
  - 動画像コーデック
  - ワープロ
  - データ検索

まとめ

- マルチコアのソフトウェアに関して
  - 各種言語処理系やコンパイル技術の復習
    - 言語処理系
      - OpenMP, StreamIT, CUDA, X10, Habanero
    - 並列化コンパイル技術
      - ループ並列処理のための解析
- 早稲田大学OSCAR自動並列化コンパイラ
  - マルチグレイン並列処理、メモリ最適化、データ転送最適化、消費電力最適化
  - 情報家電向けAPI
  - 情報家電向けマルチコアRP1/RP2
  - ヘテロジニアスマルチコア