CEDEC 2009

2009年9月2日（水）

ハイパフォーマンスコンピューティングとゲームの”深いい関係”

"Close relationship" between
High Performance Computing and Computer Game

佐藤　三久

筑波大学　計算科学研究センター

概要

- 現在、超ハイエンドの世界最速のスーパーコンピュータは1ペタフロップス（一秒間に、１千兆回の浮動小数点演算）の性能に達しています。

- 実は、そこで用いられているプロセッサはPS3に使われているCellであるのは有名な話ですが、いまのHPCのトレンドは、GPUなどのゲームと関係の深い先端のテクノロジを使いこなすかが重要になってきています。

- 一方、通常のPCにおいてもマルチコアあるいはマルチプロセッサになり、その性能を生かすためには、これまでHPCで行われてきた並列プログラミングが重要になってきています。

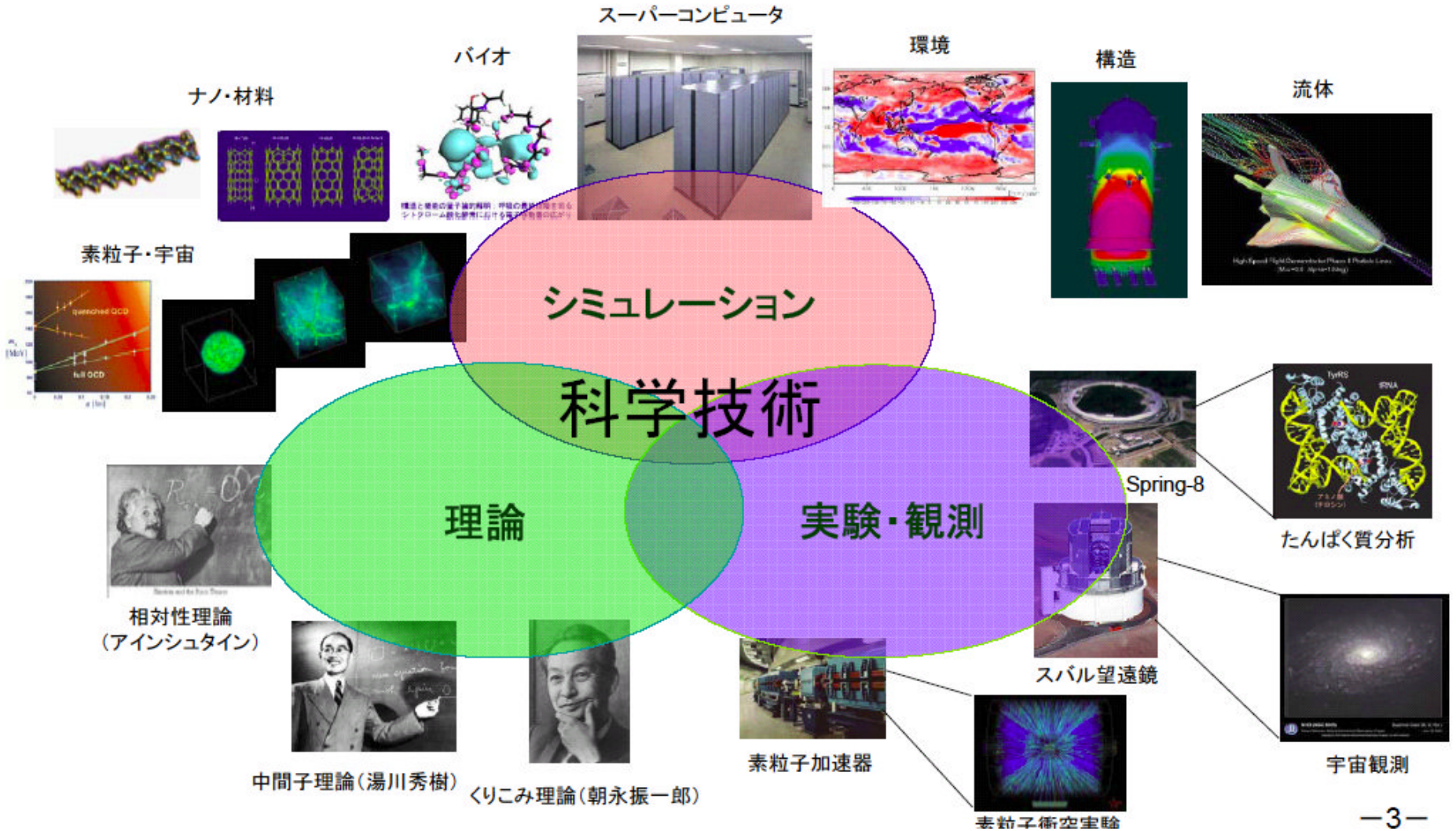
- このように関係の深い、２つのカテゴリの技術の関係に着目し、並列処理、並列プログラミング技術の概要、これからの技術のトレンドについて概観します。

目次

- HPC（ハイパフォーマンスコンピューティング）と計算科学
  - 計算科学とは
  - 計算機はどのくらい早くなったか
- 並列プログラミングについて
  - プログラミングモデル
    - OpenMP
    - メッセージ通信
    - Asynchronous RPC
  - マルチコアプロセッサの使い方
    - なぜマルチコアか　～　マイクロプロセッサのトレンド
    - マルチコアの分類　SMP　vs. AMP
- これからのHPCとゲームとの”関係”

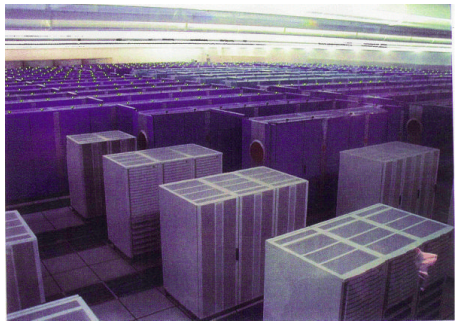
科学の三本柱としての計算科学

- 超高速計算機（スーパーコンピュータ）を用いた大規模シミュレーションを中心とした研究
- 科学技術の全分野で、実験・観測、理論と並ぶ、重要且つ最先端の研究手段

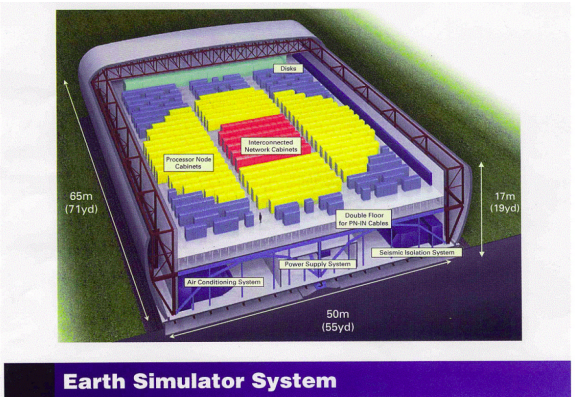


計算科学：第三の科学

- 第三の科学
  - 実験できない領域
    - 素粒子科学　Quantum chromodynamics（量子色力学）
      - クオークの質量
    - 宇宙物理学
      - 宇宙の成り立ち
    - 地球温暖化　予測
      - このまま進むとどうなるか
  - 第一原理的手法を使用すれば，実験不可能なことでも，シミュレーションによって解明される，であろうことが明らかになりつつある。
    - バイオ，ナノテクノロジー
    - 現在の計算機リソースでは不可能なものも多い
  - 実験，観測の検証。それでも実験より手軽に実施可能である。
    - 流体シミュレーション、構造設計
      - 自動車、ビル設計、飛行機・ロケットの設計



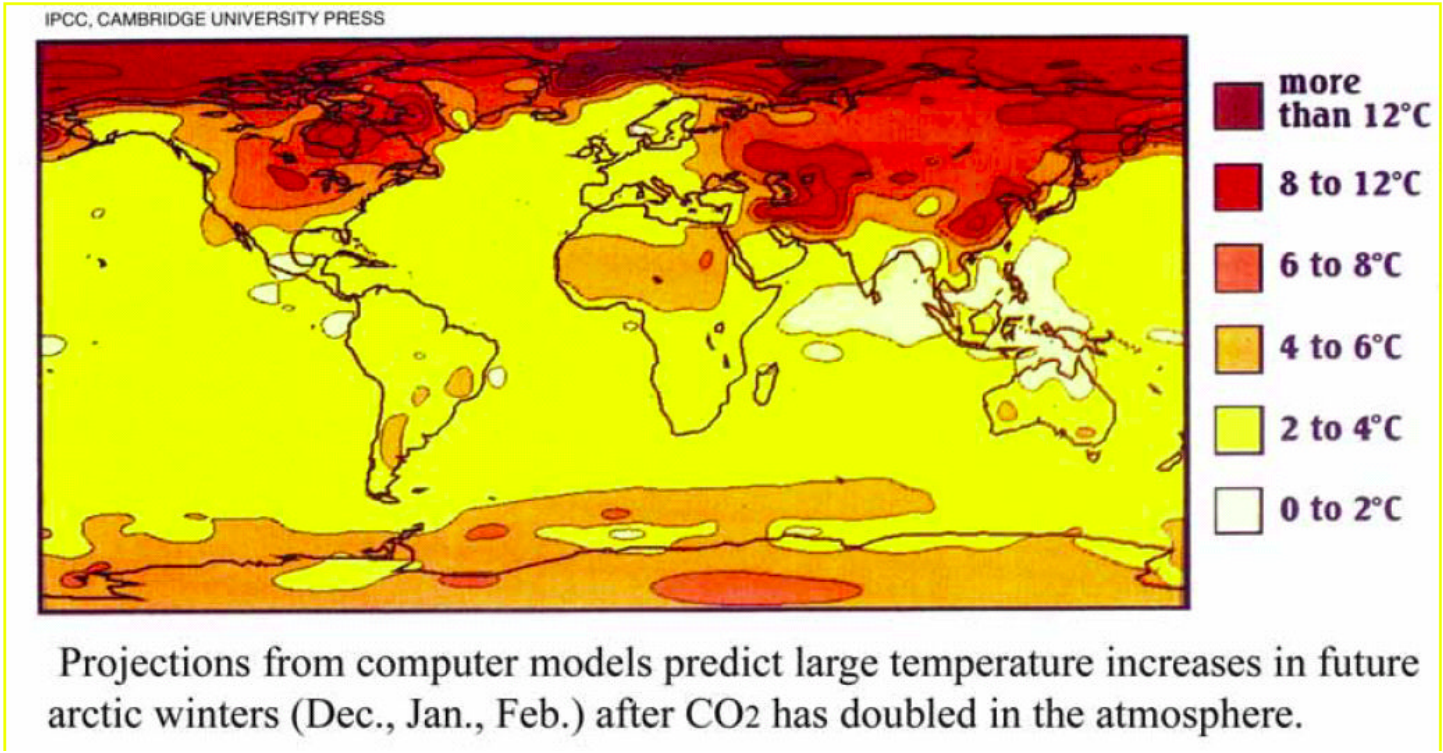
地球シミュレータ



- 海洋技術研究所・地球シミュレータセンター
- NEC
- 2002年完成
- 国産ベクトル計算機として世界最高速
- 大規模気象シミュレーション等様々な分野で応用
- 共有メモリ結合されたベクトルプロセッサ
- 5120 CPU

CO2倍増実験

大気中の二酸化炭素増加を想定したコンピューターの気候予測は
特に冬期の北極圏の気温上昇を予測している。



Projections from computer models predict large temperature increases in future arctic winters (Dec., Jan., Feb.) after CO_2 has doubled in the atmosphere.

性能　40TFlops
地球環境問題
大気循環モデル
細かくシミュレーション
20kmメッシュ、

計算機はどのくらい早くなったか

- 能力はどうやって計るのか？
  - １秒あたりの演算可能回数
  - Top500　－　Linpack(密行列の大規模連立一次方程式の求解)

- マイクロプロセッサの発展
  - クロックスピードにほぼ比例して早くなる
  - 2000年前後にクロックは1GHzに。
  - しかし、２～3GHzで、クロック速度の進歩はとまった。

- スーパーコンピュータは並列処理の時代へ
  - 単一プロセッサでは限界！
  - スーパーコンピュータは並列処理により早くなっている

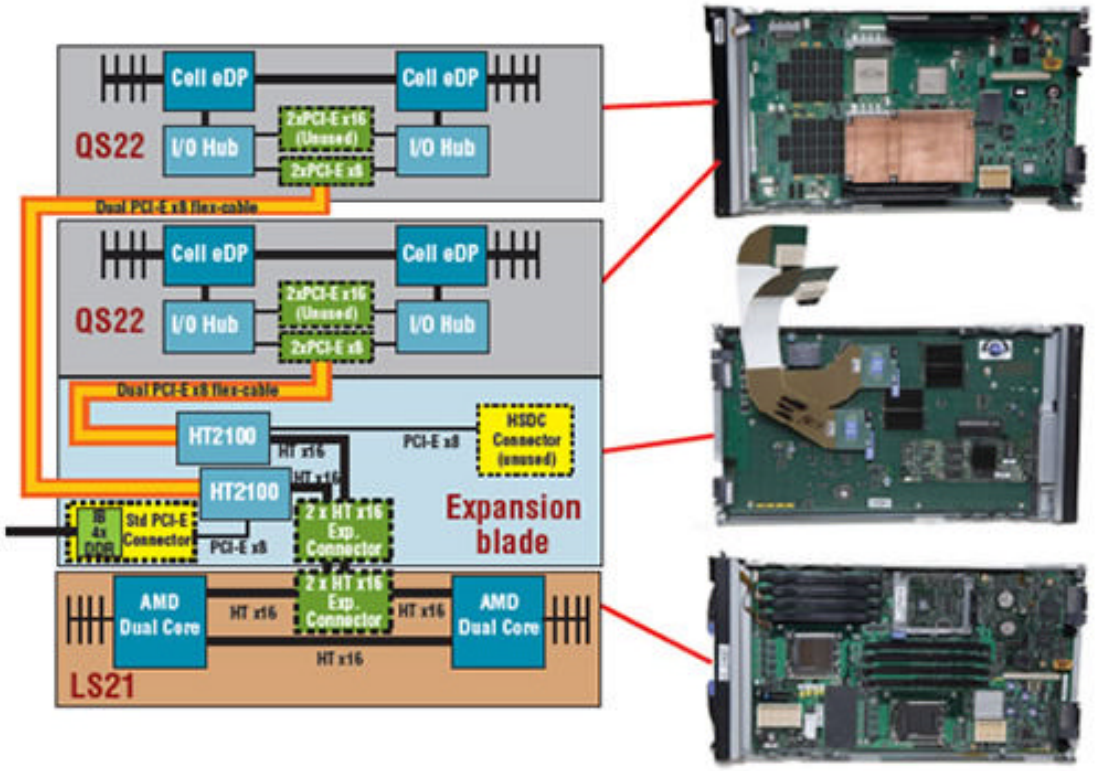
TOP 500 List
http://www.top500.org/

- LINPACKと言われるベンチマークテストを実施する。
  - 密行列を係数とする連立一次方程式を解く
  - ベクトル機でもスカラー機でも性能が出やすい
  - 例えば地球シミュレータはピーク性能40TFLOPS（設計値）に対して35 TFLOPS以上の性能が出ている。
- 実際のアプリケーションではこれほどの性能は出ない
  - 差分法，スペクトル法系の手法：ピーク性能の60%程度
    - AFES on the Earth Simulator： 26 TFLOPS（ピーク性能の65%）
- 2008年から、電力消費量を表示するようになった
  - これからのスパコンは、電力が大切

Top500 (2008/June)

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz , Voltaire Infiniband / 2008 IBM	122400	1026.00	1375.78	2345.50
2	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution / 2007 IBM	212992	478.20	596.38	2329.60
3	Argonne National Laboratory United States	Blue Gene/P Solution / 2007 IBM	163840	450.30	557.06	1260.00
4	Texas Advanced Computing Center/Univ. of Texas United States	Ranger - SunBlade x6420, Opteron Quad 2Ghz, Infiniband / 2008 Sun Microsystems	62976	326.00	503.81	2000.00
5	DOE/Oak Ridge National Laboratory United States	Jaguar - Cray XT4 QuadCore 2.1 GHz / 2008 Cray Inc.	30976	205.00	260.20	1580.71
6	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2007 IBM	65536	180.00	222.82	504.00
7	New Mexico Computing Applications Center (NMCAC) United States	Encanto - SGI Altix ICE 8200, Xeon quad core 3.0 GHz / 2007 SGI	14336	133.20	172.03	861.63
8	Computational Research Laboratories, TATA SONS India	EKA - Cluster Platform 3000 BL460c, Xeon 53xx 3GHz, Infiniband / 2008 Hewlett-Packard	14384	132.80	172.61	786.00
9	IDRIS France	Blue Gene/P Solution / 2008 IBM	40960	112.50	139.26	315.00

Roadrunner

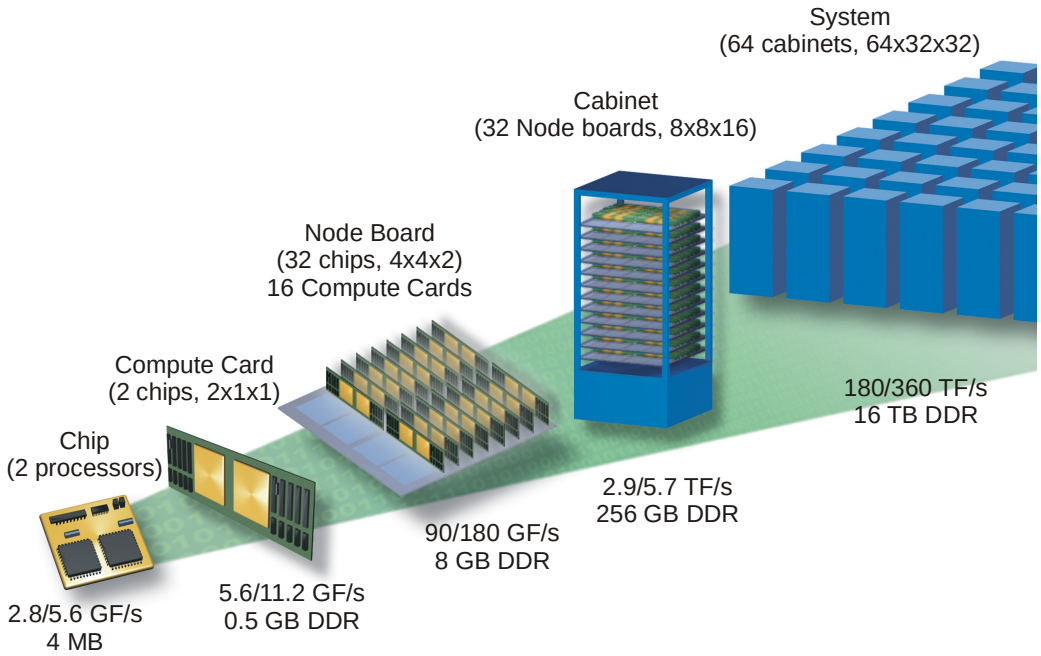


- Los Alamos National Lab.
- IBM
- 2008年完成
- 各ノードにOpteronプロセッサとIBM Cell Broadband Engineを搭載したハイブリッド型クラスタ
- 世界初のPFLOPSコンピュータ
  ⇒ 2008/6-11, 2009/6の3期連続 TOP500#1
- 129600プロセッサ
  1.46 PFLOPS
  (Linpack: 1.11 PFLOPS)

Blue Gene/L

- Lawrence Livermore National Lab.
- IBM
- 2005年完成
- 組み込み用低性能プロセッサを非常に多数ネットワーク結合
- 素粒子計算、流体計算等
- 65536 CPU
  360 TFLOPS

PowerPC440組込CPU改



TSUBAME

- 東京工業大学
- SUN Microsystems + NEC
- 2006年完成
- 各計算ノードをmulti-core CPU (dual-core Opteron) とアクセラレータ(ClearSpeed) によって構成したハイブリッドクラスタ
- 計算科学全般
- 655 nodes/10480 cores
- 109.7TFLOPS(アクセラレータ込み)

T2K-Tsukuba

- 筑波大学計算科学研究センター.
- Appro International + Cray Japan
- 2008年完成（6月稼動開始）
- ノード性能とネットワーク性能をコモディティとして最高レベルに上げたPCクラスタ
- 計算科学全般
- 2592 CPU chip = 10368 CPU core
  95 TFLOPS

計算機はどのくらい早くなったか

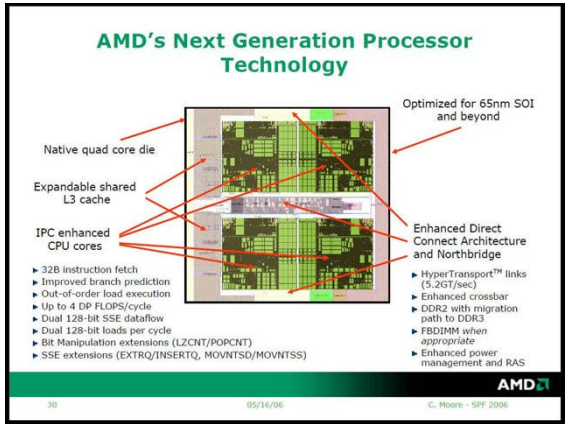
- 能力はどうやって計るのか？
  - １秒あたりの演算可能回数
  - Top500　－　Linpack(密行列の大規模連立一次方程式の求解)

- マイクロプロセッサの発展
  - クロックスピードにほぼ比例して早くなる
  - 2000年前後にクロックは1GHzに。
  - しかし、２～3GHzで、クロック速度の進歩はとまった。

- スーパーコンピュータは並列処理の時代へ
  - 単一プロセッサでは限界！
  - スーパーコンピュータは並列処理により早くなっている

プロセッサ研究開発の動向

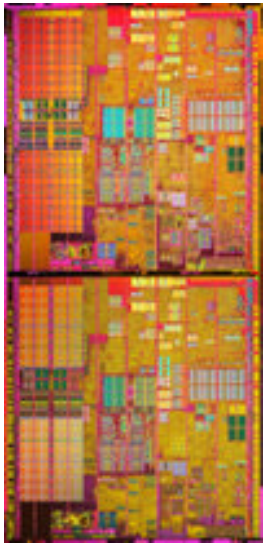
- クロックの高速化、製造プロセスの微細化
  - いまでは3GHz, 数年のうちに10GHzか！？
    - インテルの戦略の転換　⇒　マルチコア
    - クロックは早くならない！？
  - プロセスは65nm⇒45nm , 将来的には32nm
    - トランジスタ数は増える！



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

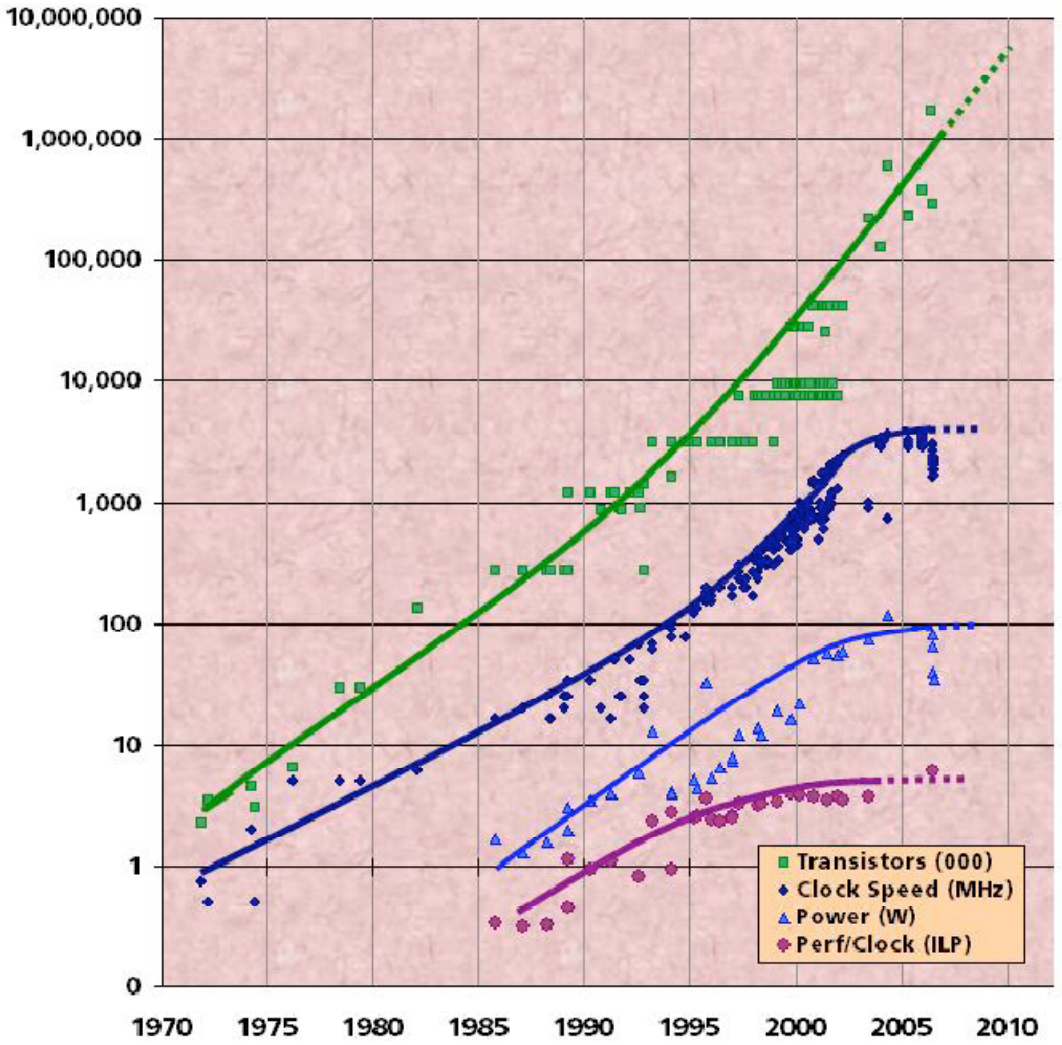
プログラミングに関係するところ



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

Moore's Law reinterpreted

- 15年で、クロックの速度の進歩はとまった。
- トランジスタ数だけは、増加している。何につかうのか？



- マルチコア
  - 18ヶ月で、チップあたりのコア数は倍に

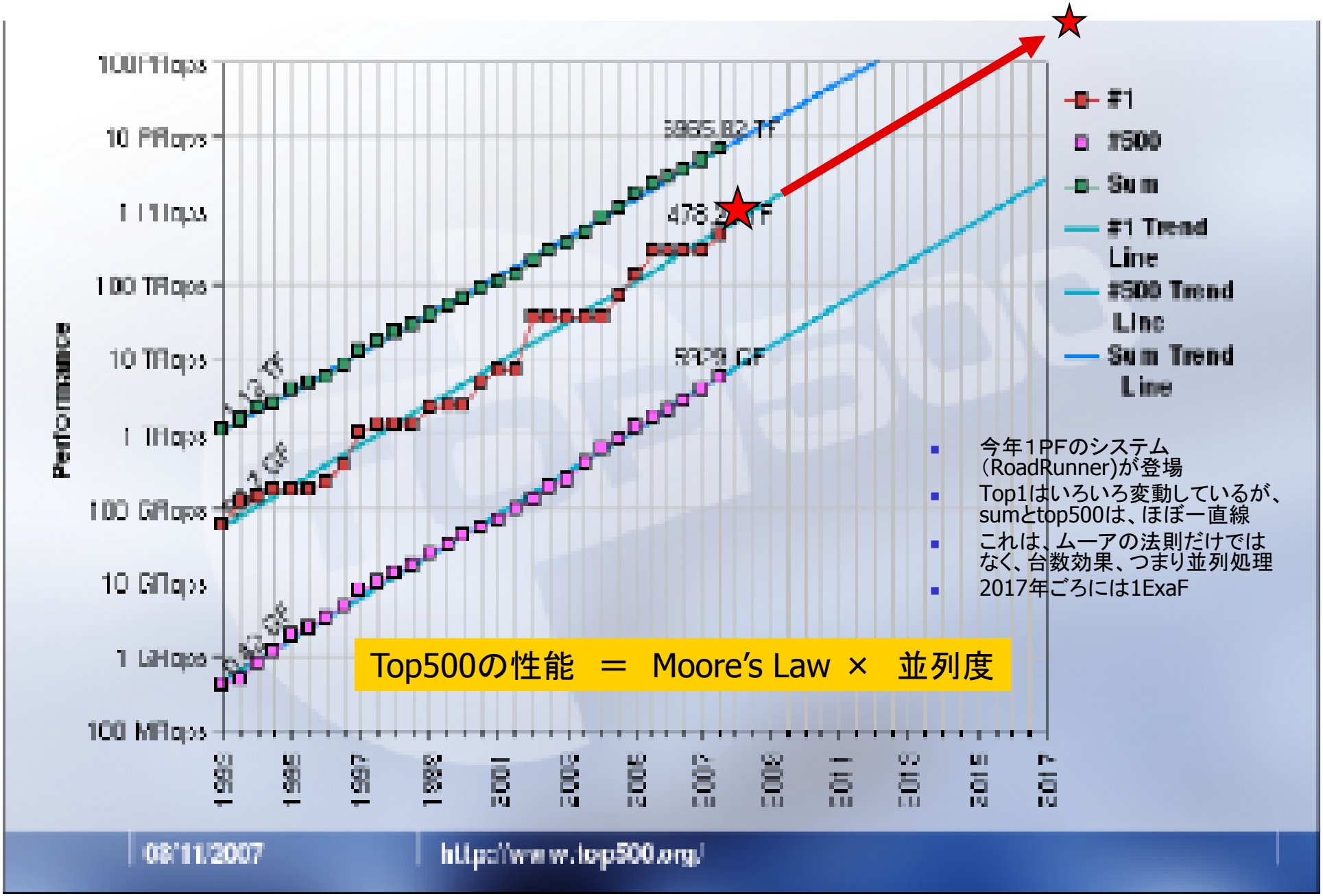
計算機はどのくらい早くなったか

- 能力はどうやって計るのか？
  - １秒あたりの演算可能回数
  - Top500　－　Linpack(密行列の大規模連立一次方程式の求解)

- マイクロプロセッサの発展
  - クロックスピードにほぼ比例して早くなる
  - 2000年前後にクロックは1GHzに。
  - しかし、２～3GHzで、クロック速度の進歩はとまった。

- スーパーコンピュータは並列処理の時代へ
  - 単一プロセッサでは限界！
  - スーパーコンピュータは並列処理により早くなっている

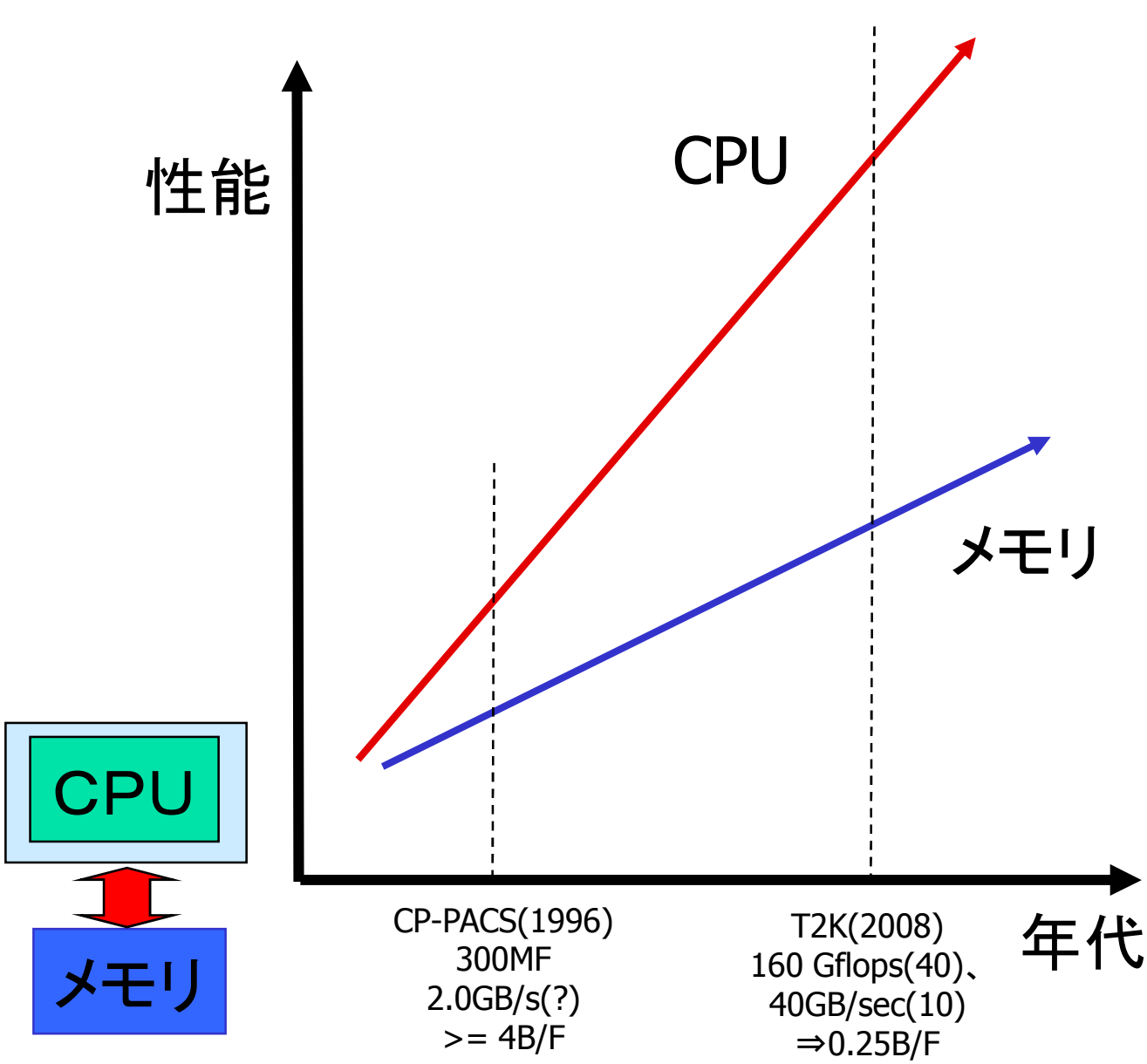
TOP500：全世界のスパコンランキング500位

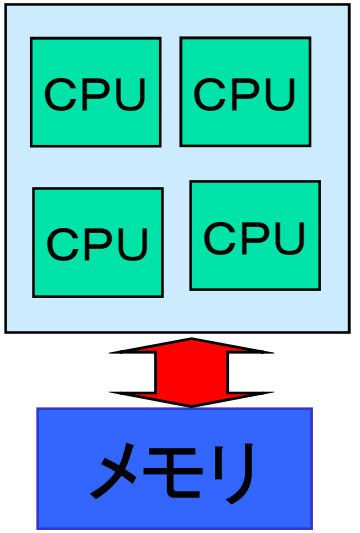


Top500の性能 ＝ Moore's Law × 並列度

- 今年1PFのシステム(RoadRunner)が登場
- Top1はいろいろ変動しているが、sumとtop500は、ほぼ一直線
- これは、ムーアの法則だけではなく、台数効果、つまり並列処理
- 2017年ごろには1ExaF

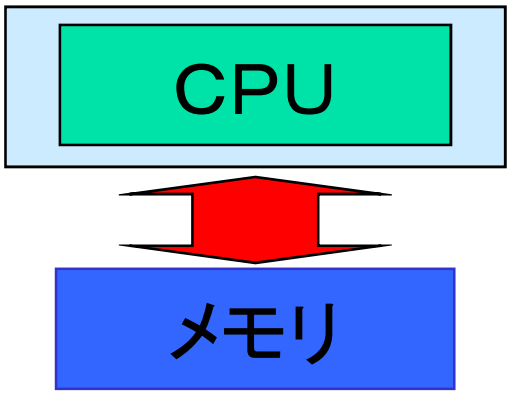
スパコン（高性能システム）のトレンド





- CPUとメモリの性能の差が広がっている
- 集積度が上がっているため
- その割にはメモリとチップのピンネックになっている
- メモリバンド幅を向上させるにはコストが高い
- それに大して、チップ側は比較的簡単に集積度を上げることができる
- ベクトルは、メモリにお金をかける分だけ高価
- メモリをつかわない（キャッシュを利用する）プログラムにする必要がある。

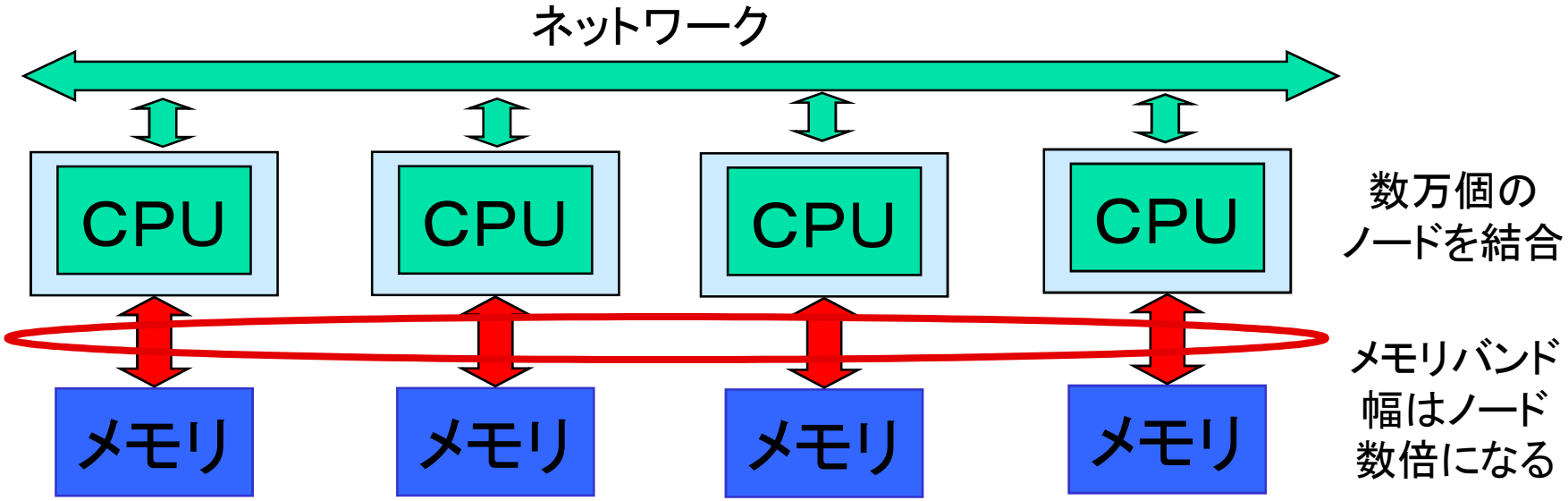
並列システムはなぜ有効か



ベクトルプロセッサで
CPUの性能を上げた
分だけ、メモリバンド
幅を上げる
⇒ しかし、高々数倍

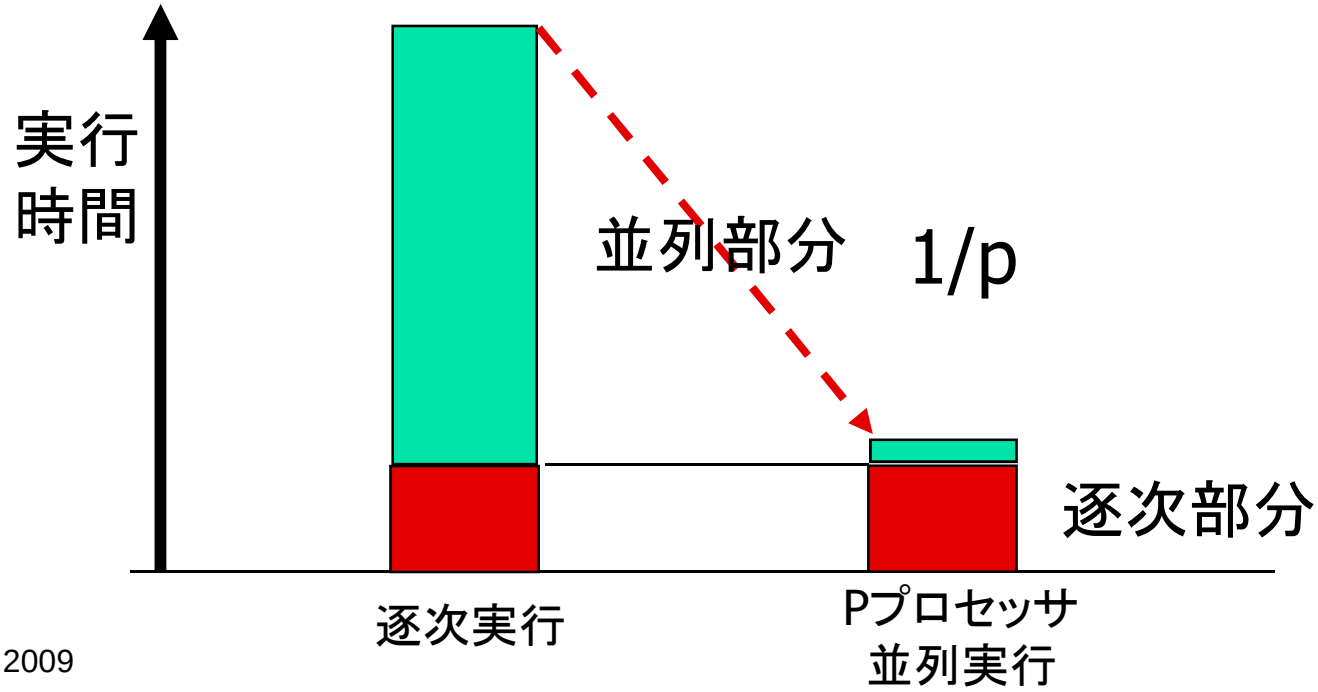
- 複数のノードを結合することにより、実行のメモリバンド幅(aggregate)を増やすことができる。
- これからは、数万ノードの時代

- もちろん、ノードの性能を引き出すことも重要
- 安いCPUのテクノロジーを利用できる。
- ネットワークの性能が重要になる。



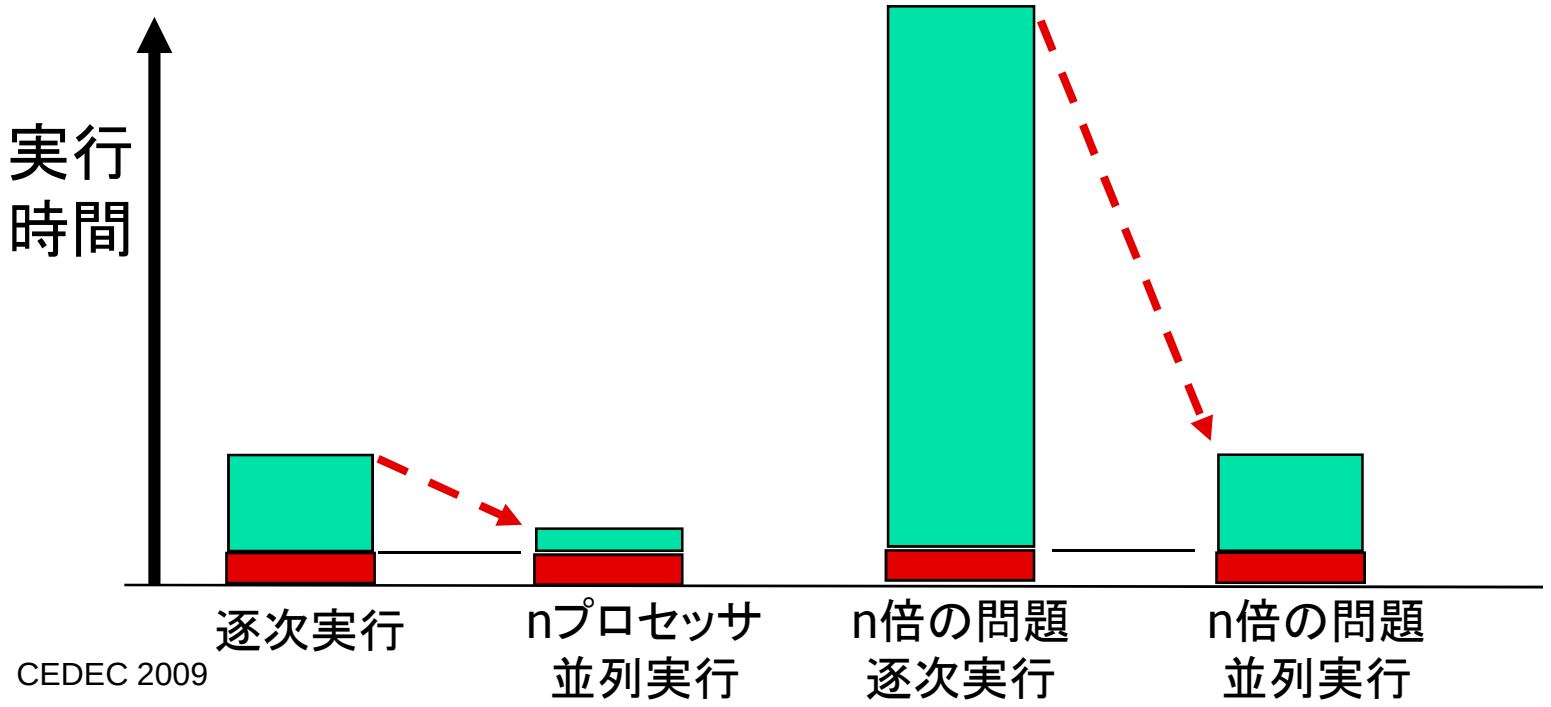
並列処理の問題点:「アムダールの法則」の呪縛

- 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、$T_p = \alpha * T_1 + (1-\alpha) * T_1/p$
  - つまり、逐次で実行しなくてはならない部分が10%でもあると、何万プロセッサを使っても、高々10倍にしかならない。



並列処理の問題点:「アムダールの法則」の呪縛

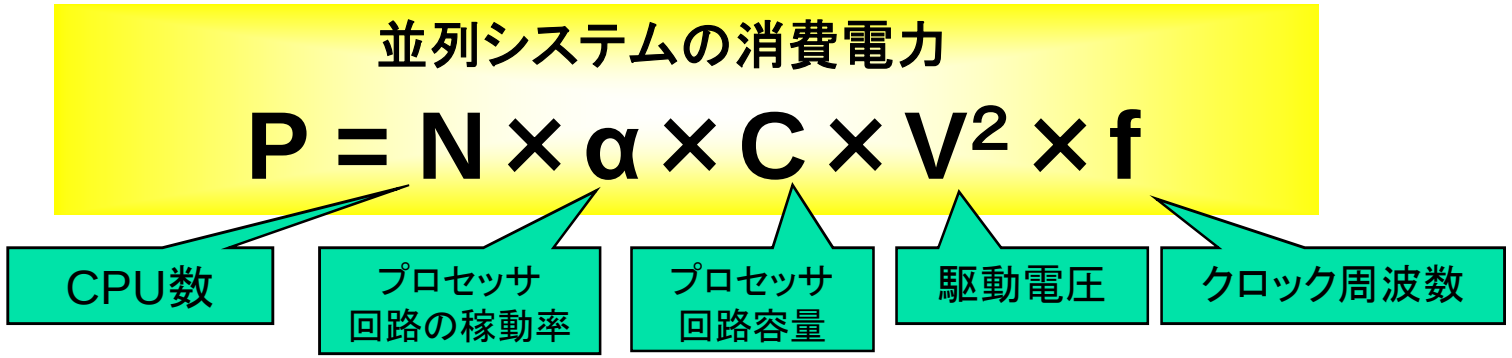
- 「Gustafsonの法則」:では実際のアプリではどうか？
  - 並列部分は問題規模によることが多い
  - 例えば、ノード数nの場合、n倍の大きい問題を解けばよい。n倍の問題は、計算量がnになると、並列処理部分は一定
  - Weak scaling – プロセッサあたりの問題を固定←大規模化は可能
  - Strong scaling －問題サイズを固定　←こちらはプロセッサが早くなくてはならない。



超ハイエンドスパコンとマルチコアによる「低電力化」

- 10PFLOPS超のシステムの実現に当たって最も重要な課題は消費電力と設置スペース
  - 現在のPC(10GFLOPS/100W)で10PFLOPSシステムを作ると、100万プロセッサで、電力は100MW必要
  - 1PFLOPSならば、現状の延長線にあるが、10PFLOPSでは再検討が必要

- マルチコアによる低電力化



並列処理による低電力化のアプローチ

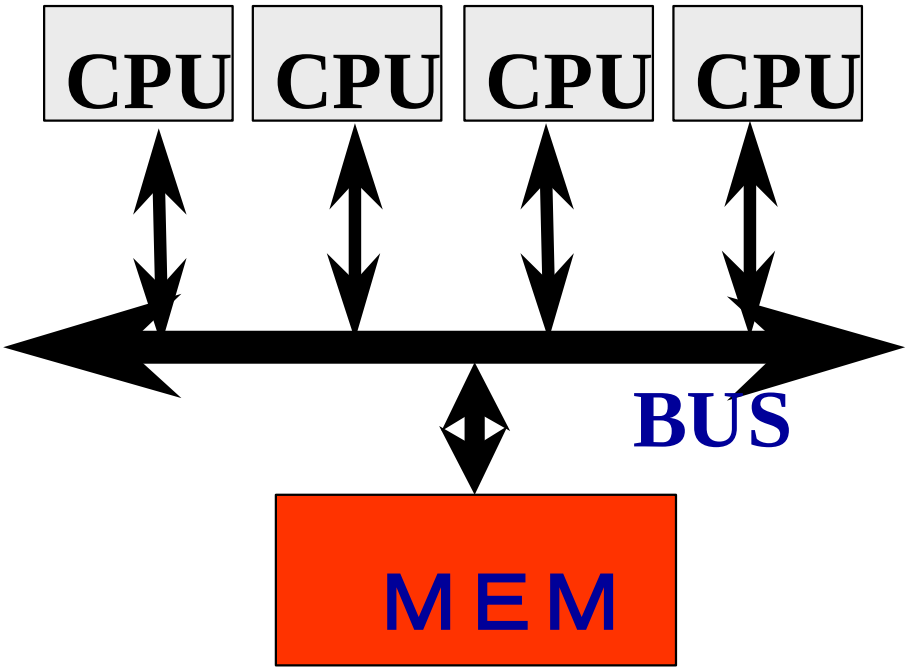
Nを増やし ↑ Vとfを押さえ ↓ システム性能 N×fを向上 ↑

- **V**と**f** を抑えることにより、チップ内の発熱密度、電力を抑える
- 微細加工技術の進歩　130nm ⇒ 90nm⇒65nm,45nm(**C**と**V**を低減)
- 低電力プロセッサ技術の活用(**α**を抑える)

HPCとゲームの”深いぃ関係”（その１）

- 今のハイエンドのスーパーコンピュータは、かつてのベクトルコンピュータとは違い、「通常の」（コモディティ）プロセッサをコンポーネントを基本としている。
  - 高性能プロセッサ、マルチコア、低電力プロセッサ、…
  - これをたくさん集めたもの
  - （ネットワーク）
- むしろ、高性能なゲームプロセッサ技術によって、生まれたコモディティ技術が、これからのハイエンドシステムを牽引する
  - RoadrunnerとCell BE
  - GPU技術から生まれたGPGPU（General Purpose GPU)

共有メモリ型



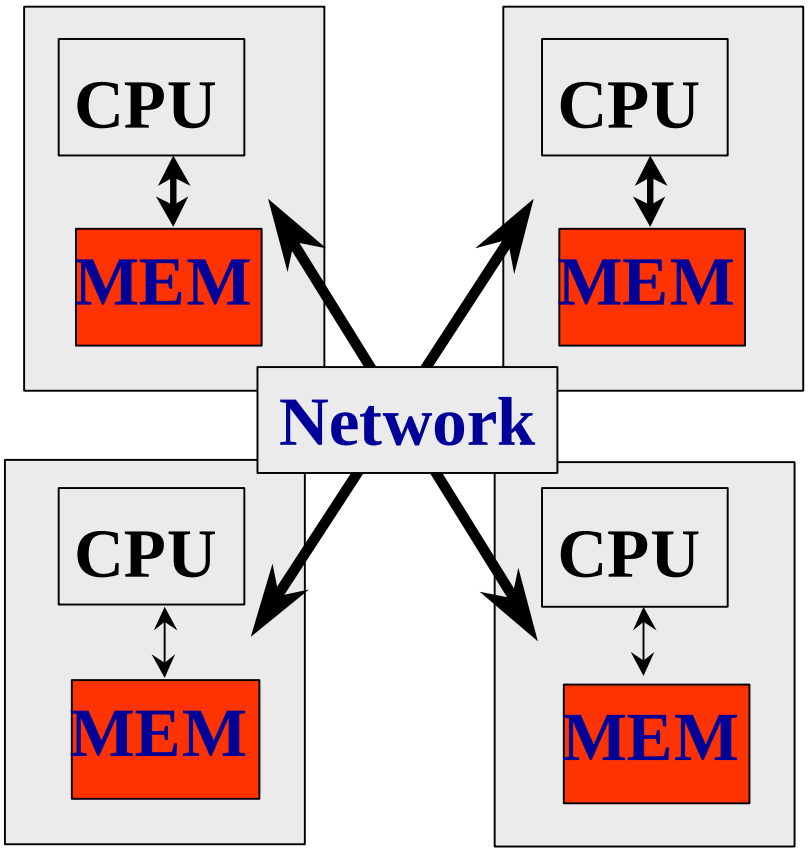
◆複数のCPUが一つのメモリにアクセスするシステム。

◆それぞれのCPUで実行されているプログラム（スレッド）は、メモリ上のデータにお互いにアクセスすることで、データを交換し、動作する。

◆大規模サーバ

◆SMPマルチコアプロセッサ

分散メモリ型



◆CPUとメモリという一つの計算機システムが、ネットワークで結合されているシステム

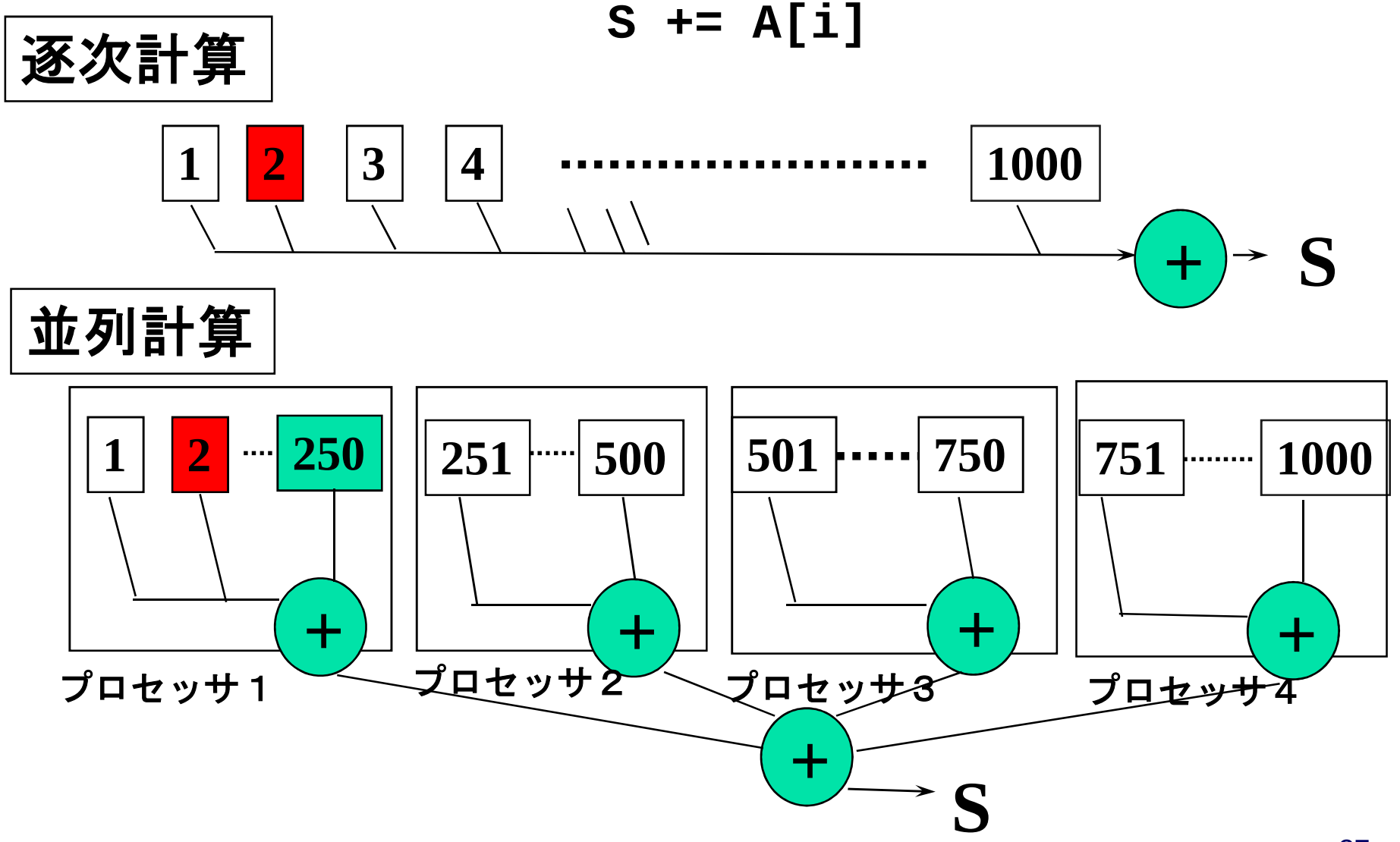
◆それぞれの計算機で実行されているプログラムはネットワークを通じて、データ(メッセージ)を交換し、動作する

◆クラスタ計算機

◆AMP　マルチコア

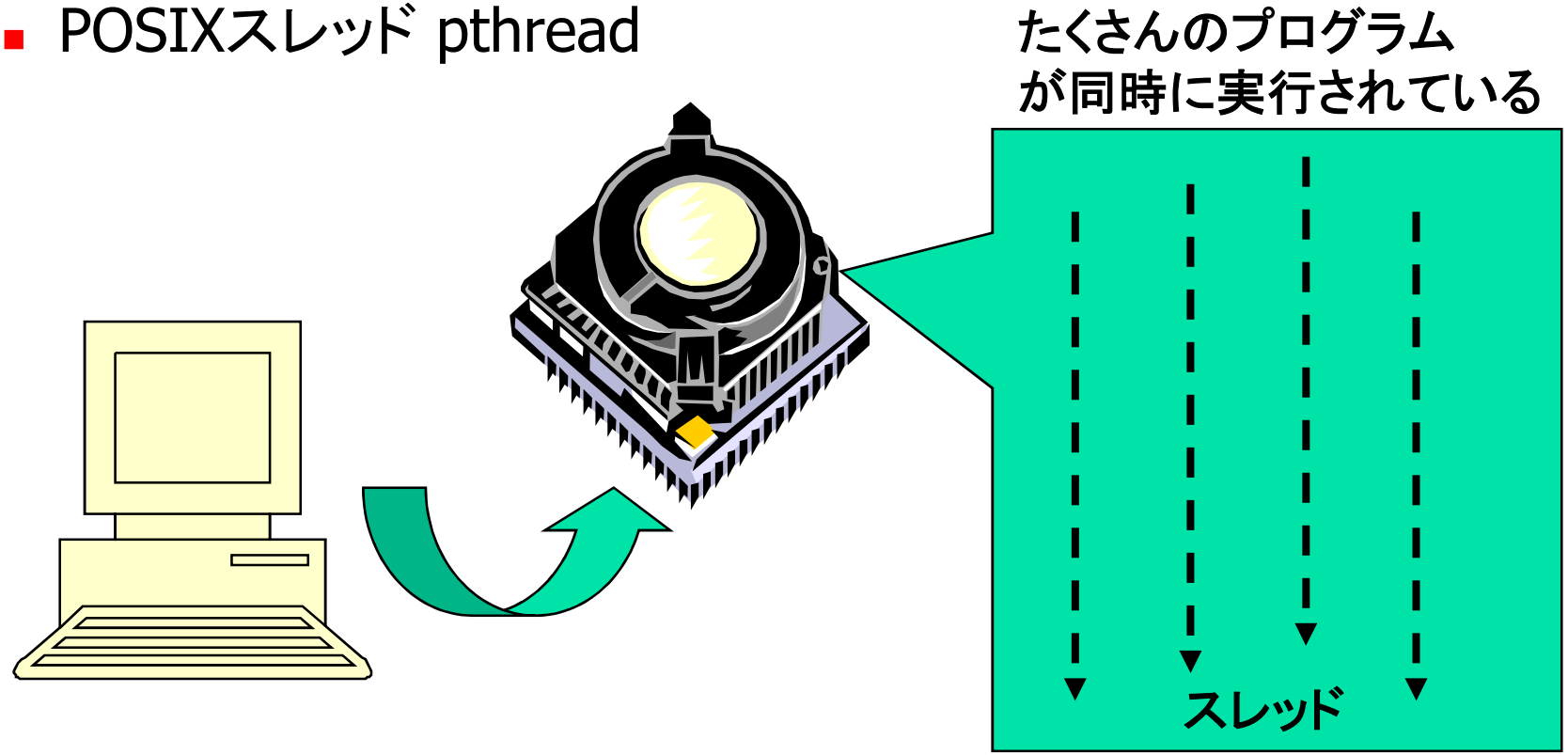
（高速化のための）並列処理の簡単な例

```
for(i=0;i<1000; i++)
    S += A[i]
```



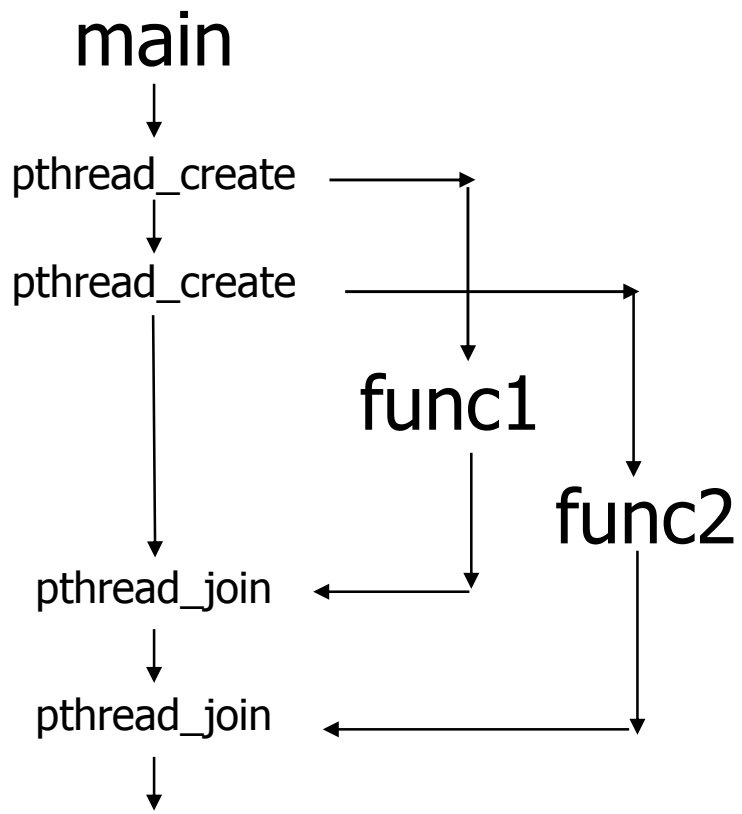
マルチスレッドプログラミング

- 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;
}
```

OpenMPによるプログラミング

これだけで、OK!

```
#pragma omp parallel for reduction(+:s)
   for(i=0; i<1000;i++) s+= a[i];
```

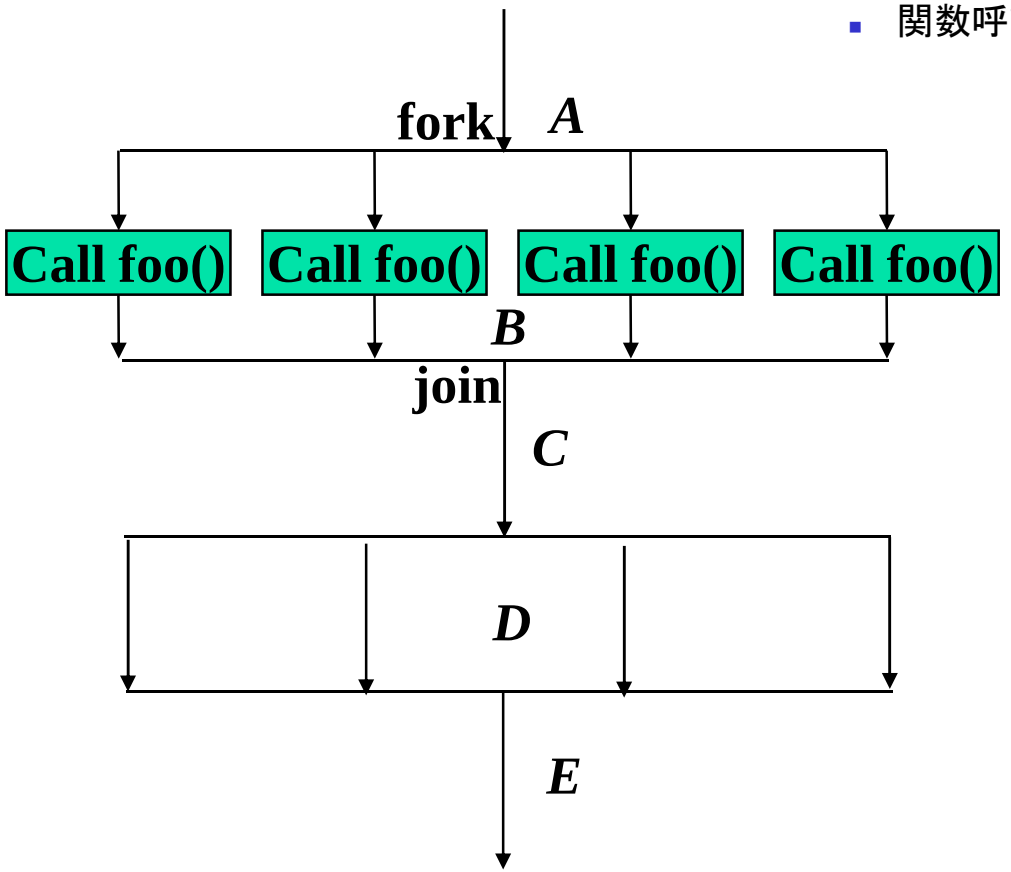
OpenMPとは http://www.openmp.org/

- 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
  - OpenMP Architecture Review Board (ARB)が仕様を決定
  - Oct. 1997 Fortran ver.1.0 API
  - 現在、OpenMP 3.0
- 新しい言語ではない！
  - コンパイラ指示文（directives/pragma）、ライブラリ、環境変数によりベース言語を拡張
  - ベース言語：Fortran77, f90, C, C++
    - Fortran：!$OMPから始まる指示行
    - C: #pragma omp のpragma指示行
- 自動並列化ではない！
  - 並列実行・同期をプログラマが明示
- 指示文を無視することにより、逐次で実行可
  - incrementalに並列化
  - プログラム開発、デバックの面から実用的
  - 逐次版と並列版を同じソースで管理ができる

OpenMPの実行モデル

- 逐次実行から始まる
- Fork-joinモデル

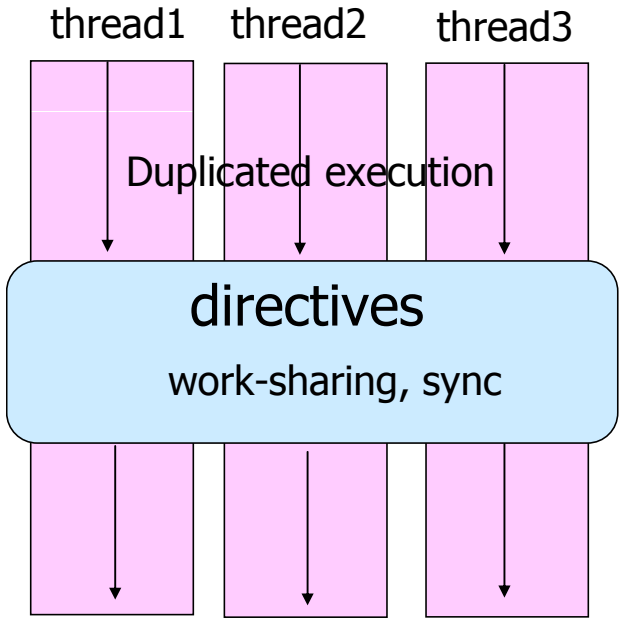
- Parallel region: 複数のスレッド(team)によって、並列実行される部分
  - Parallel構文で指定
  - 同じParallel regionを実行するスレッドをteamと呼ぶ
  - region内をteam内のスレッドで重複実行
    - 関数呼び出しも重複実行




```
… 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

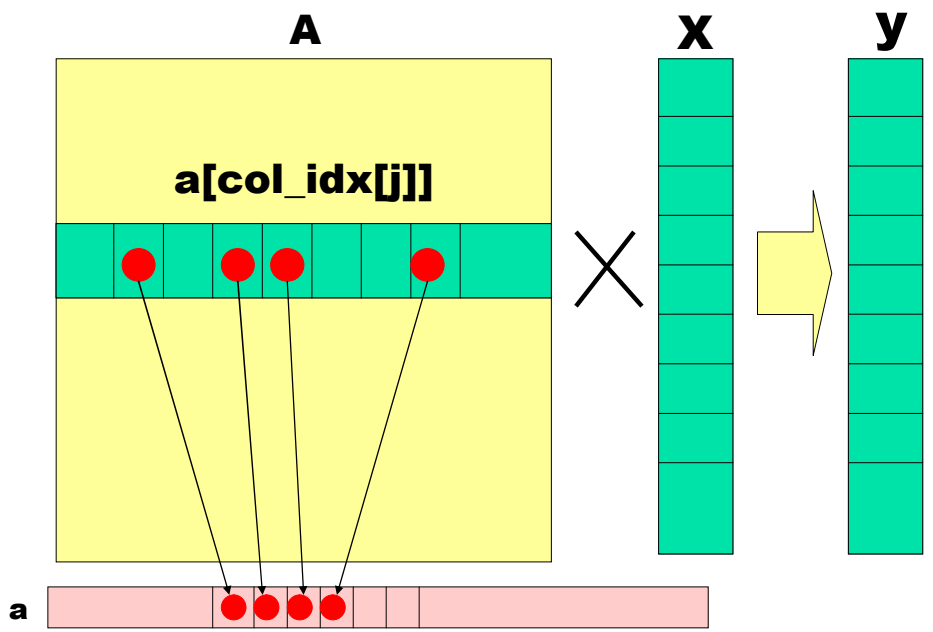
- Forループのイタレーションを並列実行
- 指示文の直後のforループは*canonical shape*でなくてはならない

```
#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;
  }
}
```



並列ループのスケジューリング

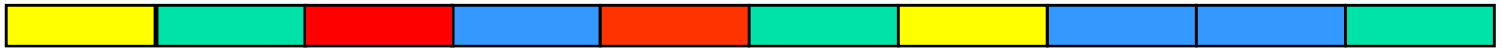
- プロセッサ数4の場合

逐次

`schedule(static,`*n*`)`

`Schedule(static)`

`Schedule(dynamic,`*n*`)`



`Schedule(guided,`*n*`)`

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

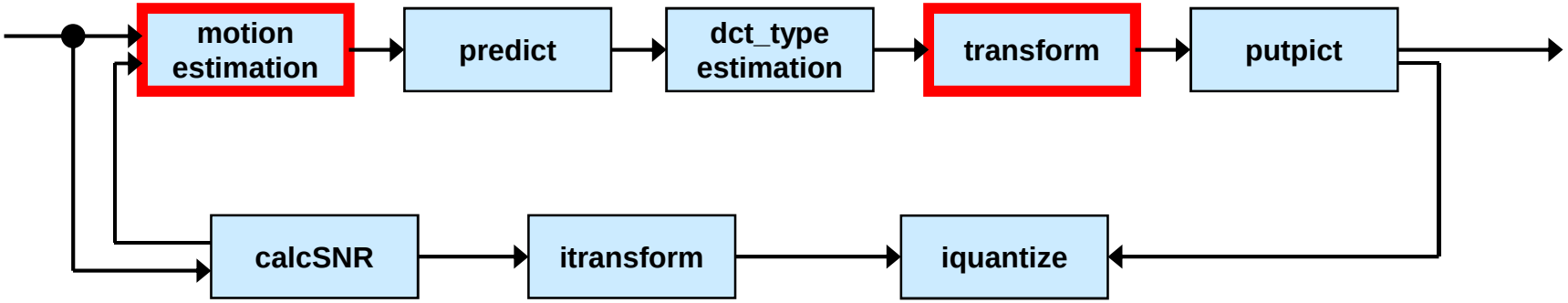
Barrier 指示文

- 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。

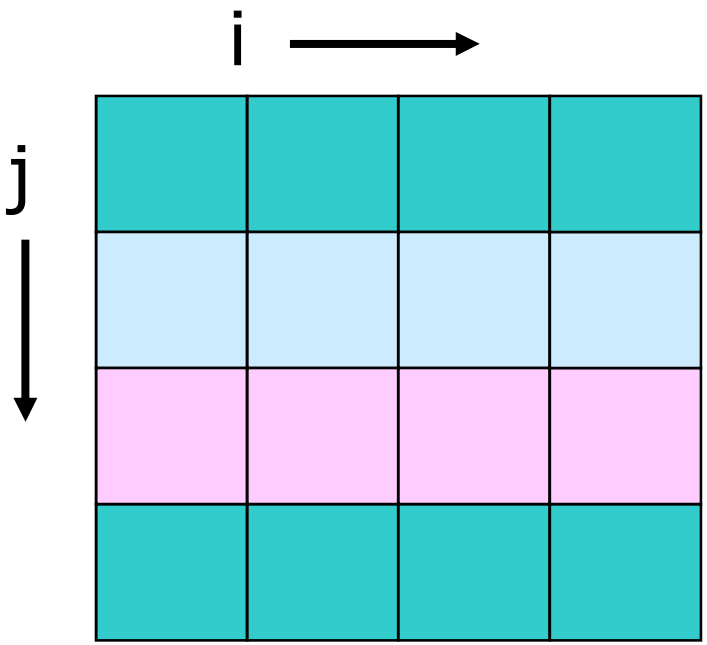
```
#pragma omp barrier
```

並列化の例：MediaBench

- MPEG2 encoder by OpenMP.

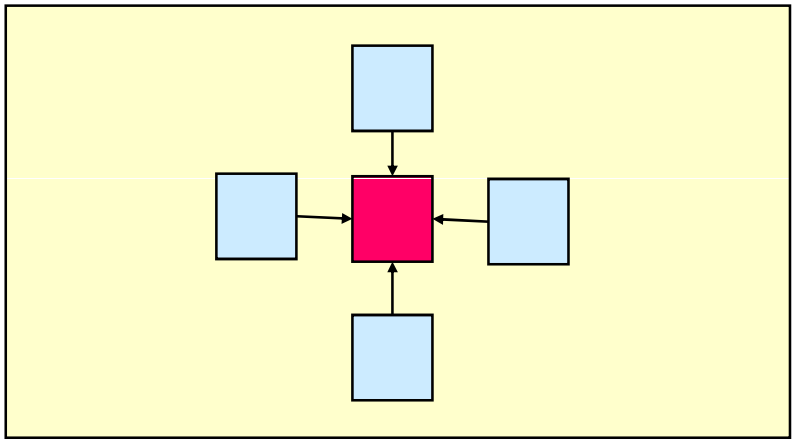



```
/*loop through all macro-blocks of the picture*/
#pragma omp parallel private(i,j,myk)
{
#pragma omp for
    for (j=0; j<height2; j+=16)
    {
        for (i=0; i<width; i+=16)
        {
            ... loop body ...
        }
    }
}
```



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
- C++

```
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);
}
```

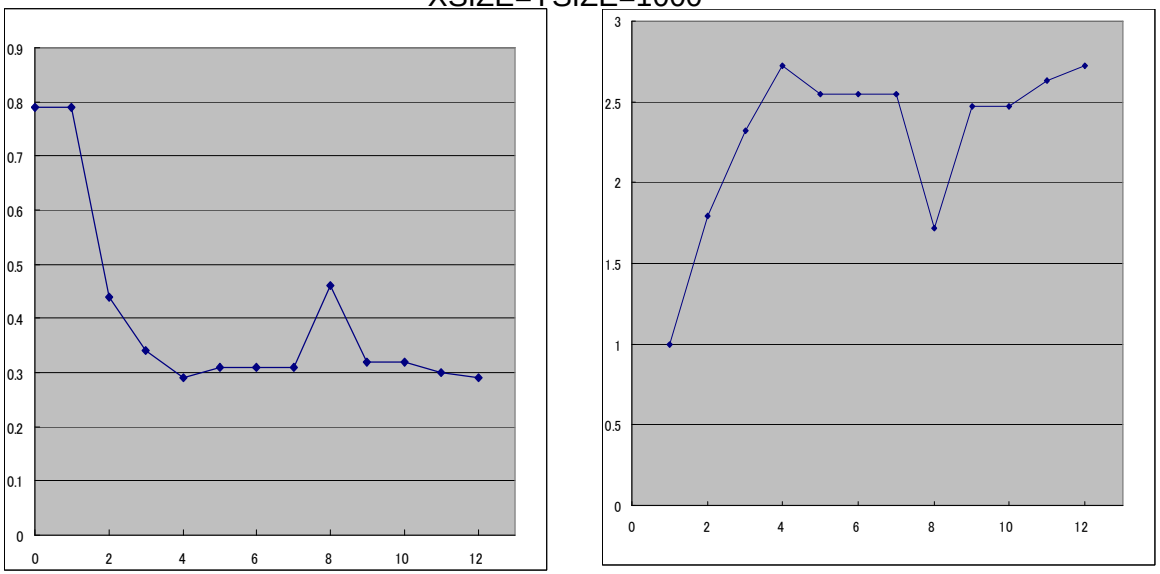
では、性能は？

- プラットフォーム、問題規模による
- 特に、問題規模は重要
  - 並列化のオーバーヘッドと並列化のgainとのトレードオフ

- 性能を得るためには…
  - ロックの方法
  - キャッシュ
  - メモリメモリバンド幅

Laplaceの性能

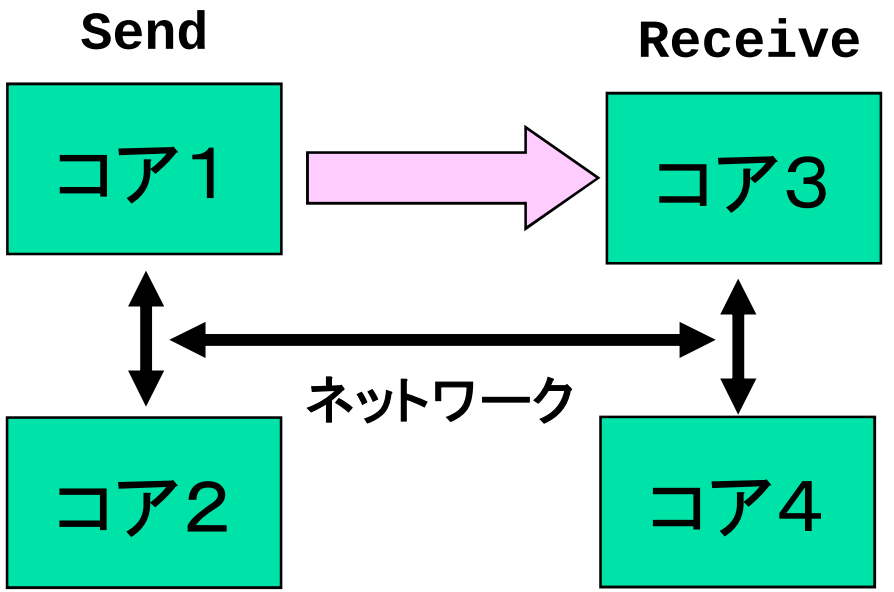
XSIZE=YSIZE=1000



Core i7 920 @ 2.67GHz, 2 socket

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

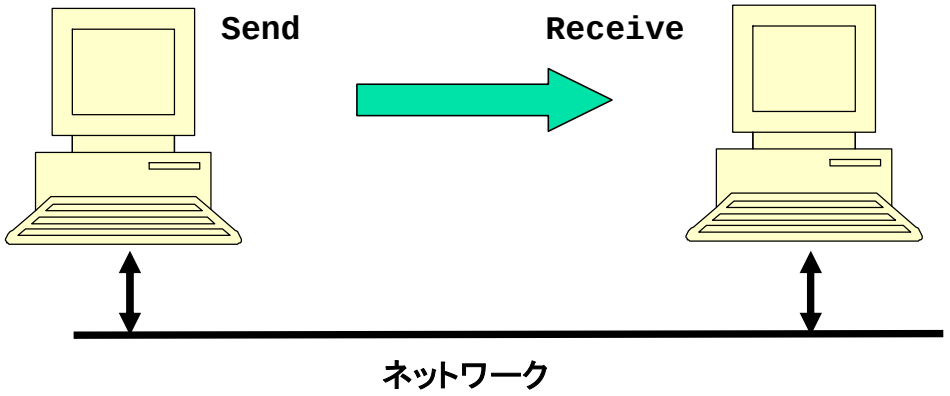
```
int a[250]; /* それぞれ、250個づづデータを持つ */

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,&status)
            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_W
     MPI_Barrier(MPI_COMM_WORLD);
    ...
    MPI_Finalize();
    return 0;
}
```

MCAPI

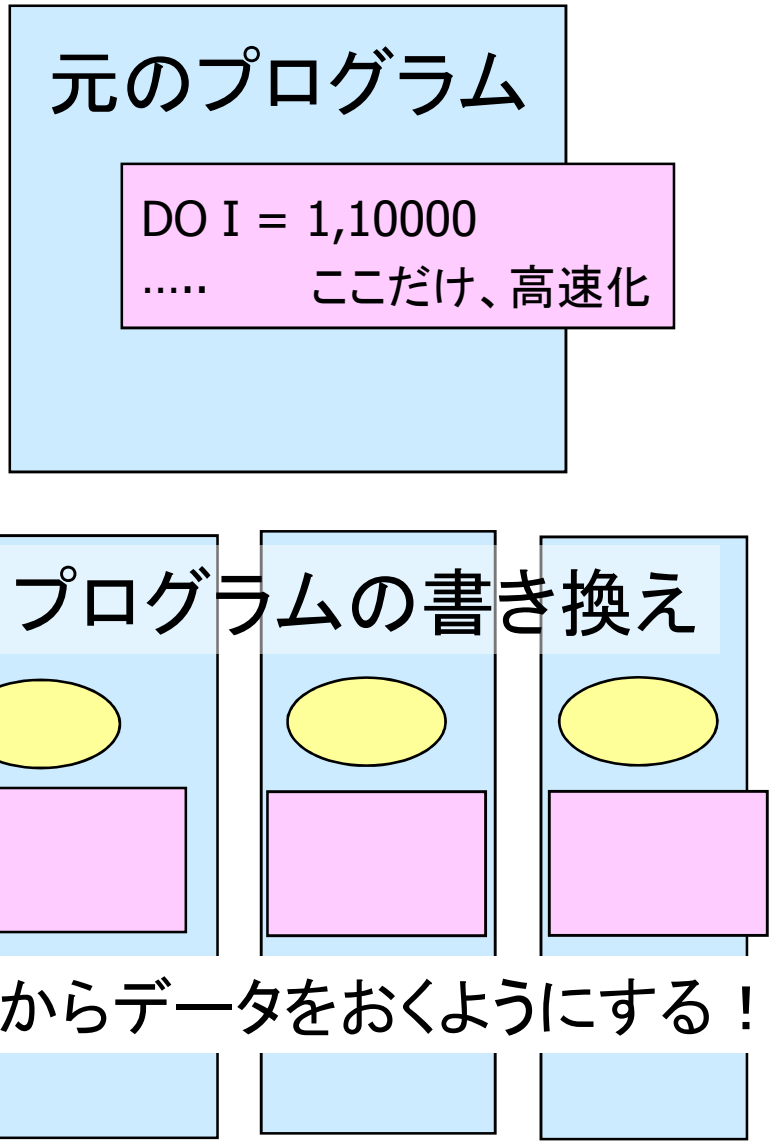
- MCAPI (Multicore Communication API)
  - Multicore Association (www.multicore-association.org, Intel, Freescale, TI, NEC)で制定された通信API
  - March 31, 2008 時点でV1.063
  - MRAPI (Resource Management API)ともに用いる
  - MPIよりも簡単、hetero, scalable, fault tolerance(?), general

- 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.

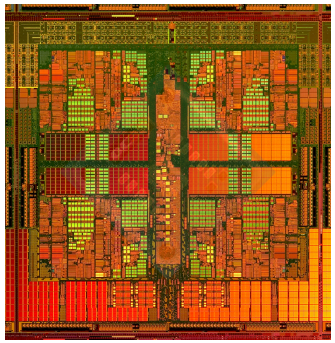
メッセージ通信の問題点：並列化はなぜ大変か

- ベクトルプロセッサ
  - あるループを依存関係がなくなるように記述
  - ローカルですむ
  - 高速化は数倍
- 並列化
  - 計算の分割だけでなく、通信（データの配置）が本質的
  - データの移動が少なくなるようにプログラムを配置
  - ライブラリ的なアプローチが取りにくい
  - 高速化は数千倍ー数万



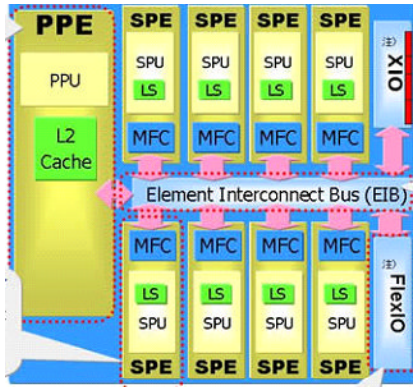
マルチコアプロセッサ：SMPとAMP

- 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

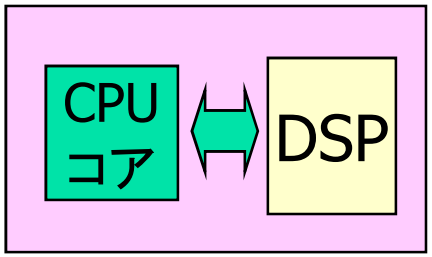


AMD quad-core

- 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



IBM Cell

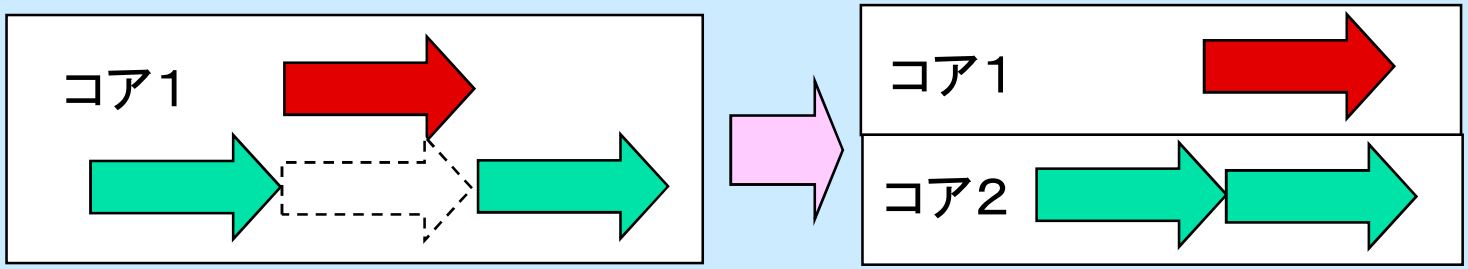


- 共有メモリ　vs. 分散メモリ
  - 各コアからどのようにメイン・メモリにアクセスできるかもプログラミングを考える場合に重要な点

マルチコアプロセッサの使い方 （その1）

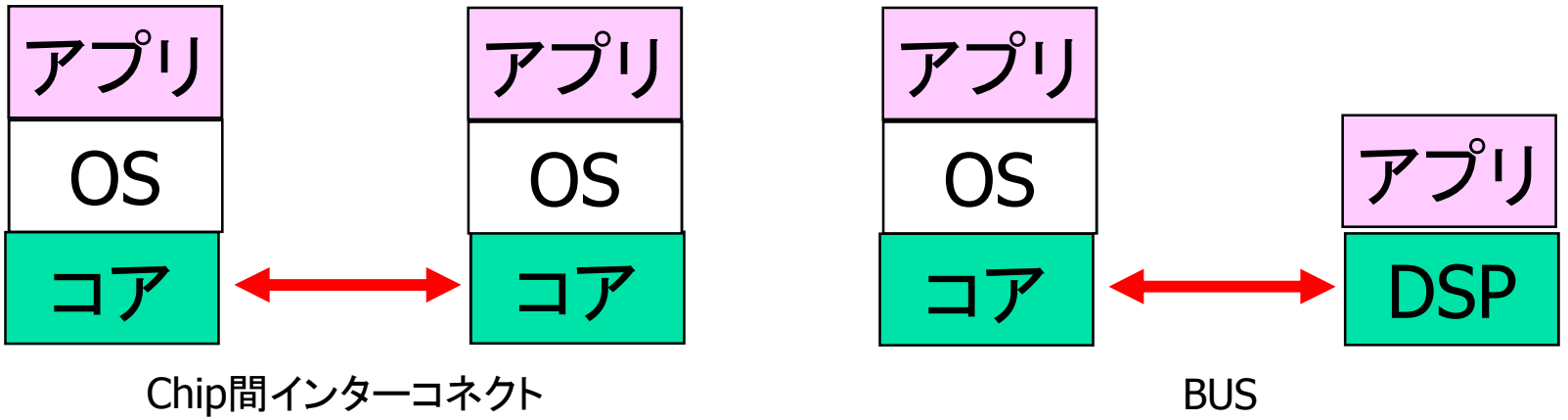
- 複数のプロセス・スレッドでの利用
  - 主に、共有メモリSMPのマルチコアプロセッサ
  - 通常、組み込みシステムはマルチタスク（プロセス）のプログラム
  - 特別なプログラミングはいらない（はず）

- 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。



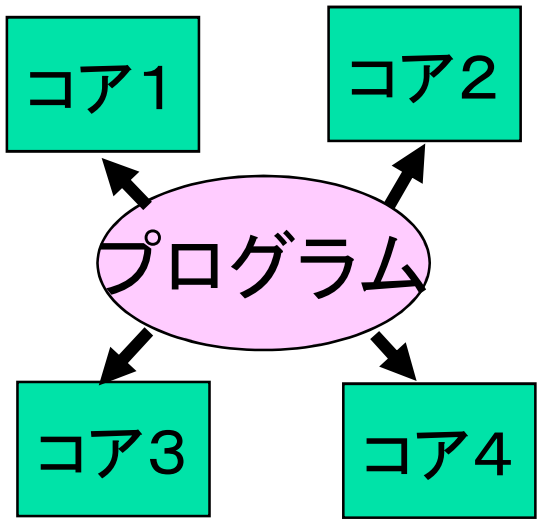
マルチコアプロセッサの使い方（その2）

- 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

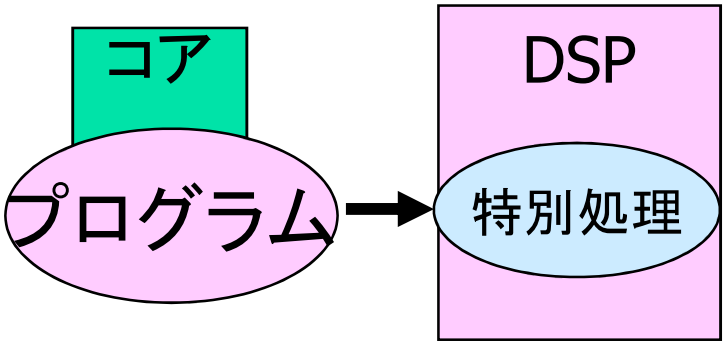


マルチコアプロセッサの使い方（3）

- 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。

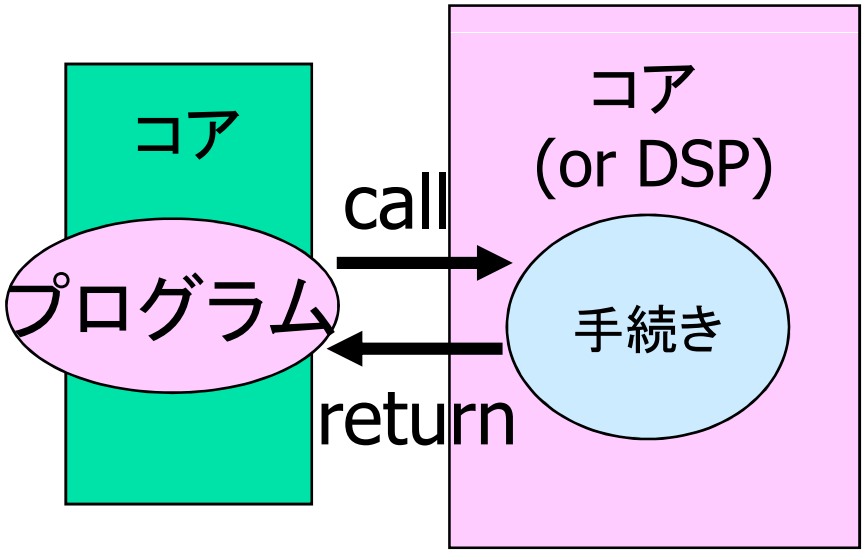


コアで分担して実行



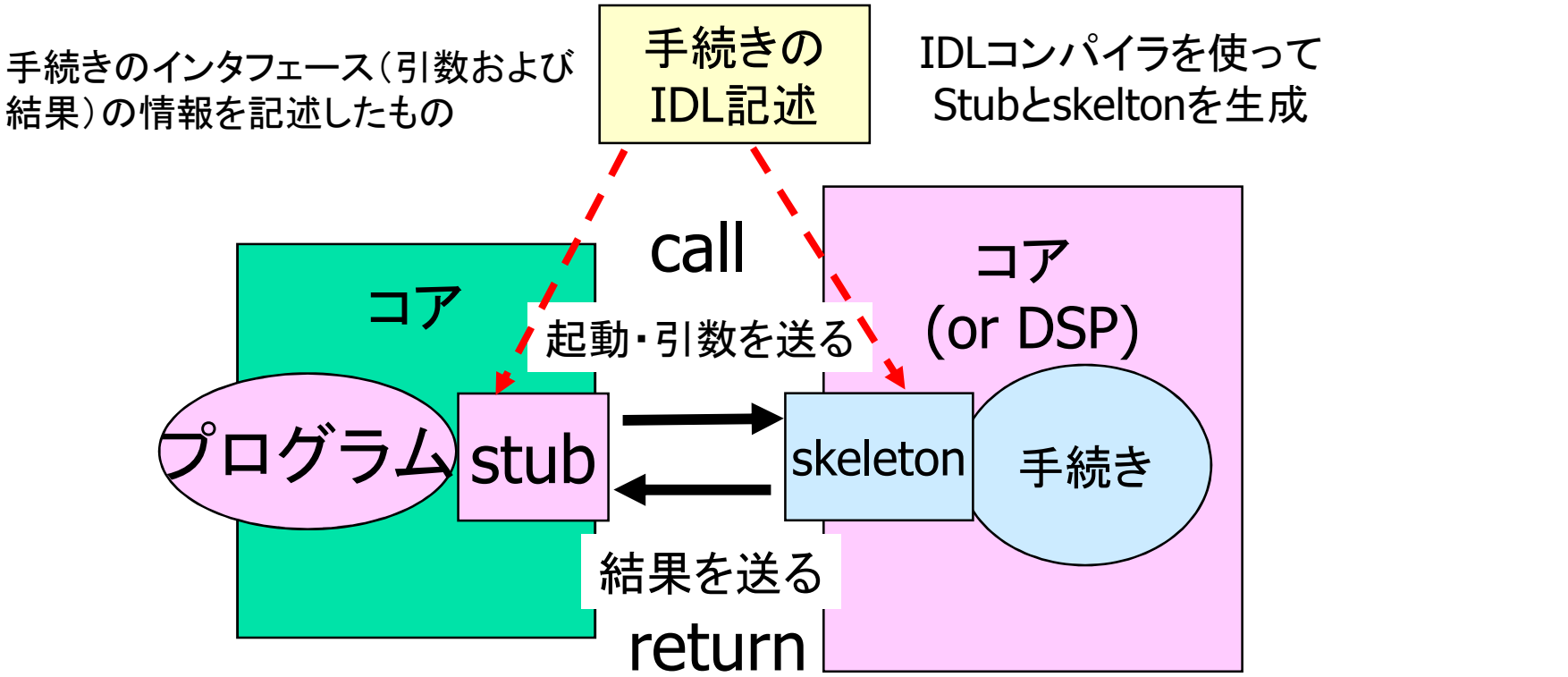
RPCによるマルチコアプログラミング

- 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



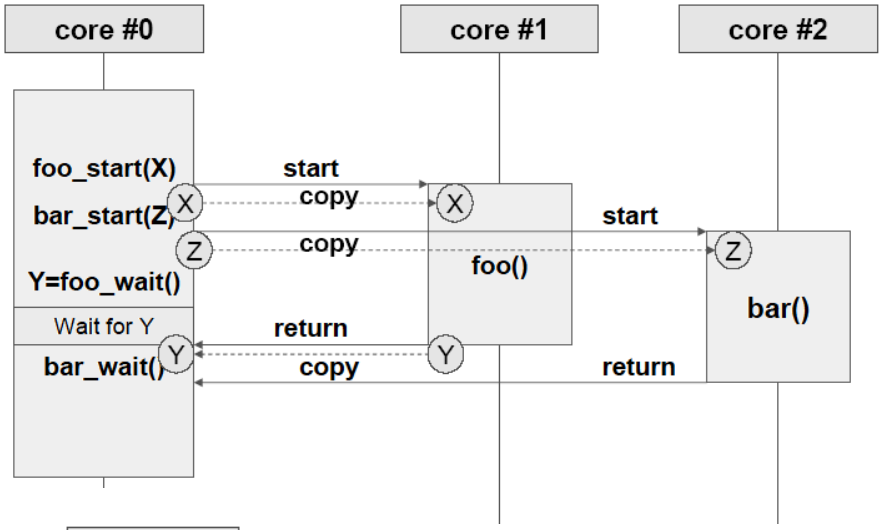
RPCの仕組み

- 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

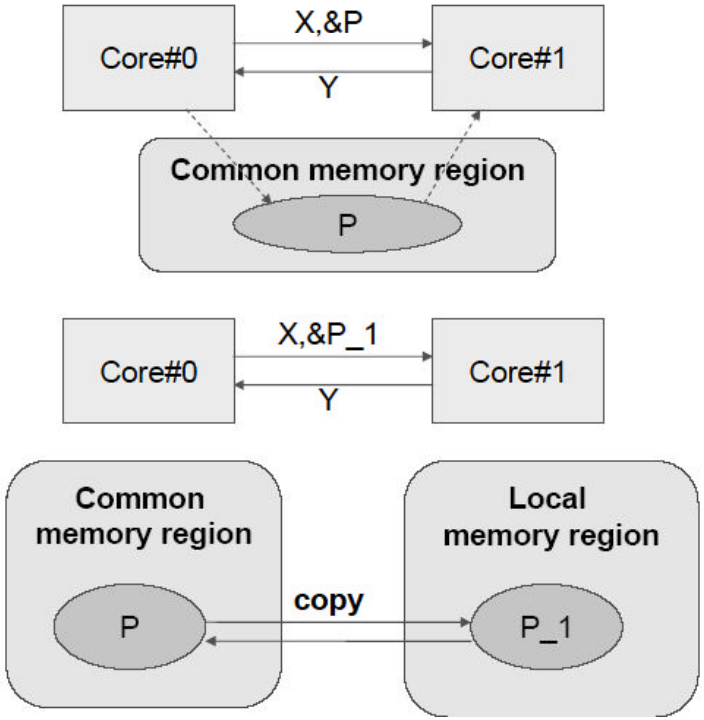


Common memory region

```
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);
```

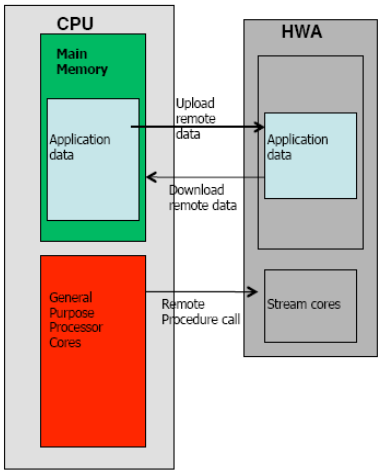


- 通常の逐次プログラムからの移行が簡単
- 通信を隠蔽することにより、いろいろなコア(&DSP)に対して移植性の良いプログラムができる

  ⇒ 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;
}
```

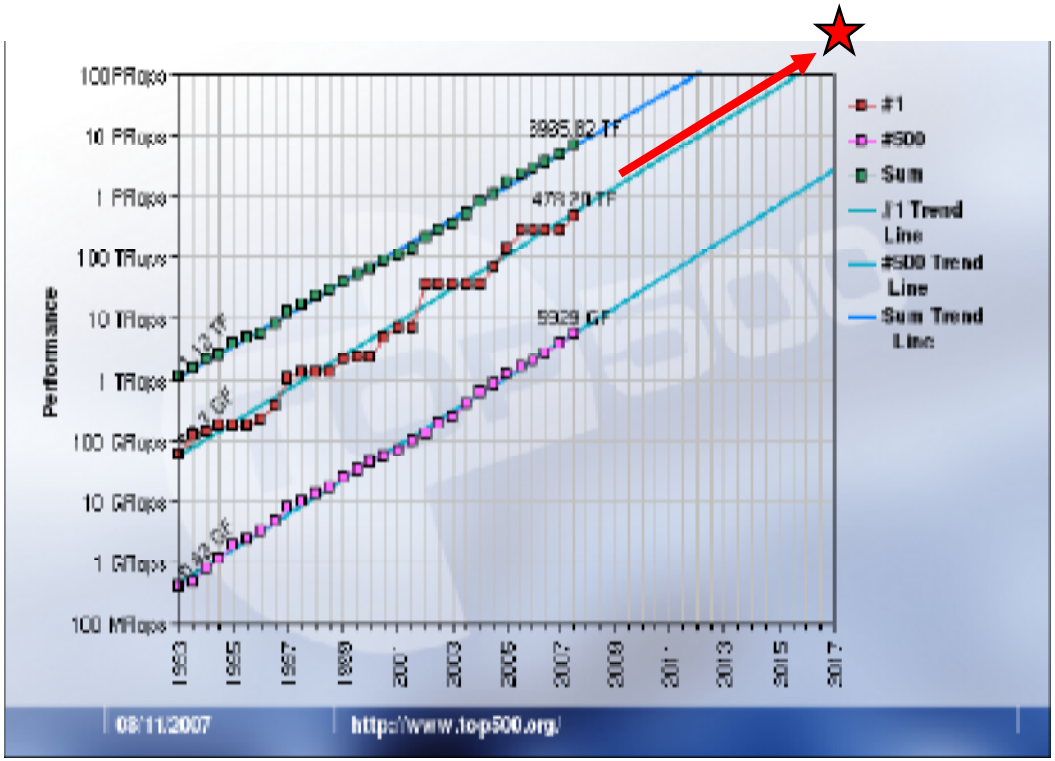
codelet / callsite directive set

HPCとゲームの”深いぃ関係”(その2)

- マルチコアとなっている、今の高性能プロセッサを使いこなすためには、HPCで培われてきた並列プログラミング技術が役に立つ
  - 共有メモリプログラミング： OpenMP
  - メッセージ通信プログラミング：MPI ⇒ MCAPI
  - RPC プログラミング
  - …
- マルチコアでも、メモリが共有されていない場合には、プログラミングが大変
- 専用組み込みプロセッサでは、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が期待されている ⇒ ARPC？OpenMP?

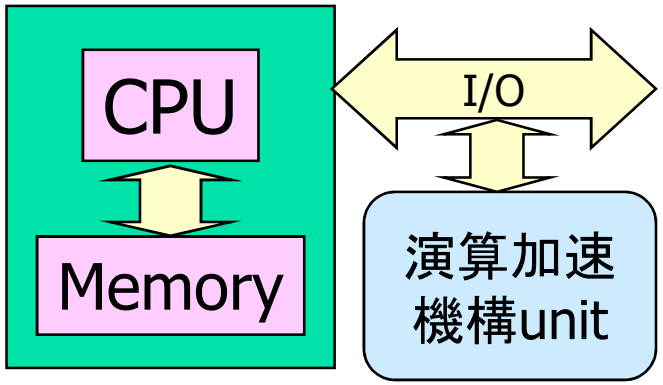
Beyond the “Petaflops”

- Approaches to “Exaflops” （10^18　百京回／秒）
  - 1,000PFlops@2017±2 (predicted by Top500)
  - これまでは、台数（ノード数）を増やすことによって、スピードを達成してきた。（weak scaling …）
  - これからはコア数（演算器）を増やす(strong scaling)ことしかできない。
  - ２つの大きな問題：
    - Power constraints
    - Memory wall
      (efficiency)



Strong Scalingのための演算加速機構

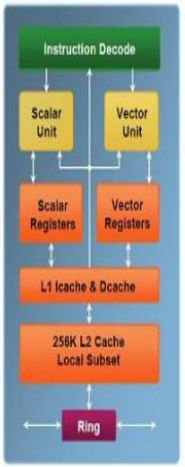
- 演算を別のUnit(ボード)にoff-loadする
  - 演算に特化した、専用な
    アーキテクチャが可能
    - nVIDIAのシェダープロセッサ
  - 専用のメモリアーキテクチャを使うことができる
    - GDDR3



- 実際の例
  - GPGPU (General-Purpose Graphic Processing Unit)
    - GPUをグラフィックだけでなく、数値計算などの一般の演算に使えるように拡張したもので、非常に活発に研究が行われている。
    - CUDAが提案されたことで、プログラミングが、簡単になった
    - nVIDIA GeForce, Tesla,…
    - いまでは、OpenMPでの並列化も提案されている(PGI OpenMP)
  - Cell BE
  - ClearSpeed
  - Intel Larrabee (many core)

代表的な演算加速機構

Larrabee x86 Core Block Diagram



- Separate scalar and vector units with separate registers
- In-order x86 scalar core
- Vector unit: 16 32-bit ops/clock
- Short execution pipelines
- Fast access from L1 cache
- Direct connection to each core's subset of the L2 cache
- Prefetch instructions load L1 and L2 caches

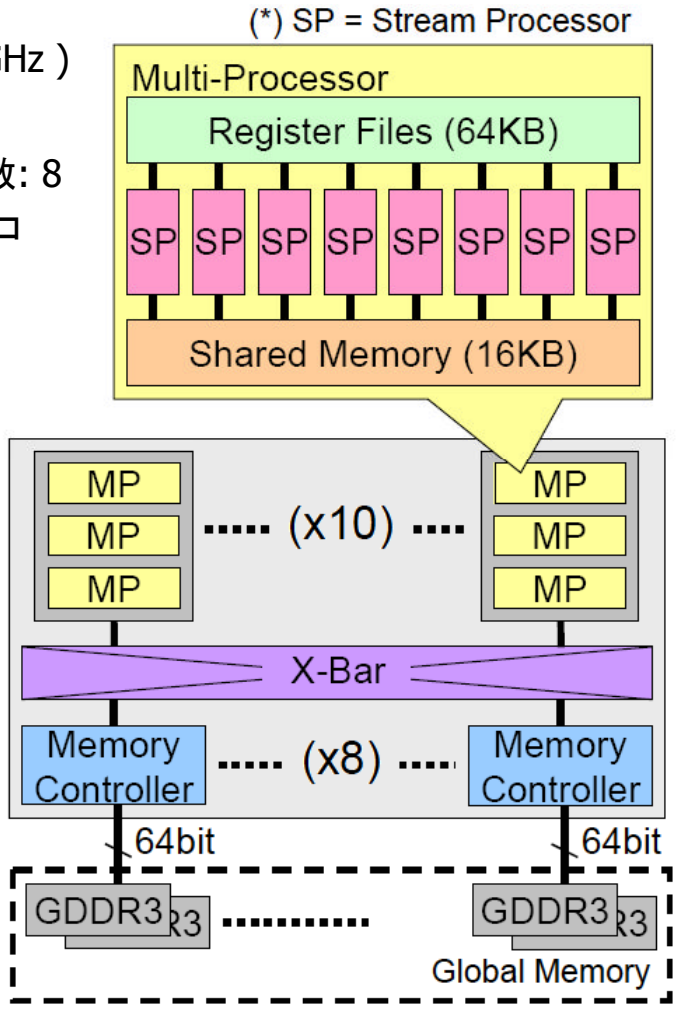
Larrabee Block Diagram

- Cores communicate on a wide ring bus
  - Fast access to memory and fixed function blocks
  - Fast access for cache coherency
- L2 cache is partitioned among the cores
  - Provides high aggregate bandwidth
  - Allows data replication & sharing

CEDEC 2009

- 理論ピークメモリバンド幅
  - 141.7GB/s
  (= 64bit * 8 * 2.214GHz)
- MP数: 30
  - MPの内部構成• SP数: 8
  (全体で240シェダープロセッサ)
  - 993GFlops
  - 共有メモリ: 16KB
  - レジスタ数: 16K本 (64KB)

GeForce GTX280



富士通研究所　成瀬氏のスライドから

http://news.cnet.com/8301-13512_3-10006184-23.htm より

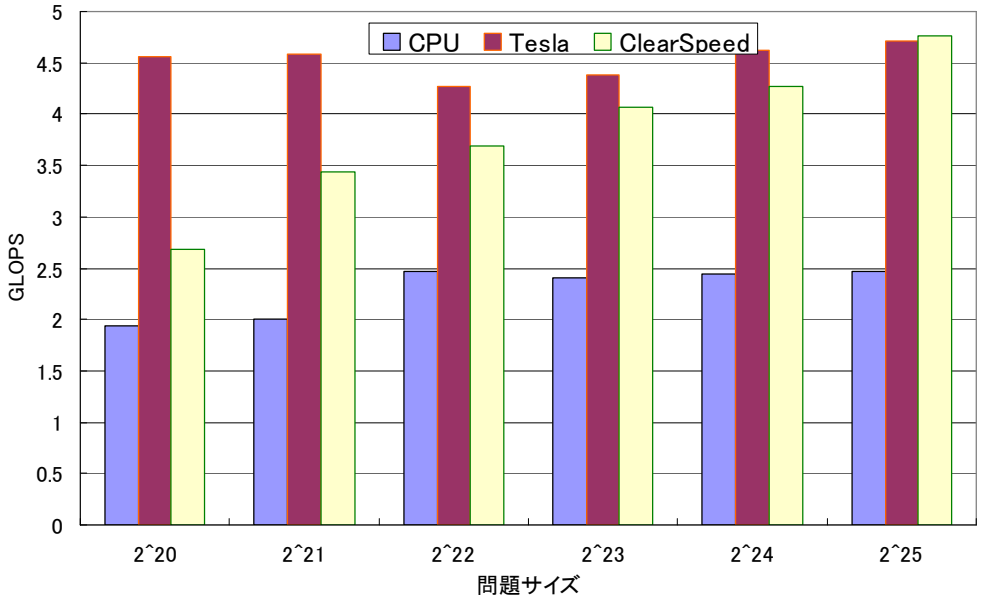
性能は？

- ClearSpeed Advance e620
  （以下 ClearSpeedと表記）
- NVIDIA TeslaC870
  (以下 Teslaと表記)
- CPU:AMD Quad-Core Opteron 2.0GHz × 2
  (メモリ：DDR2-800 16GB)

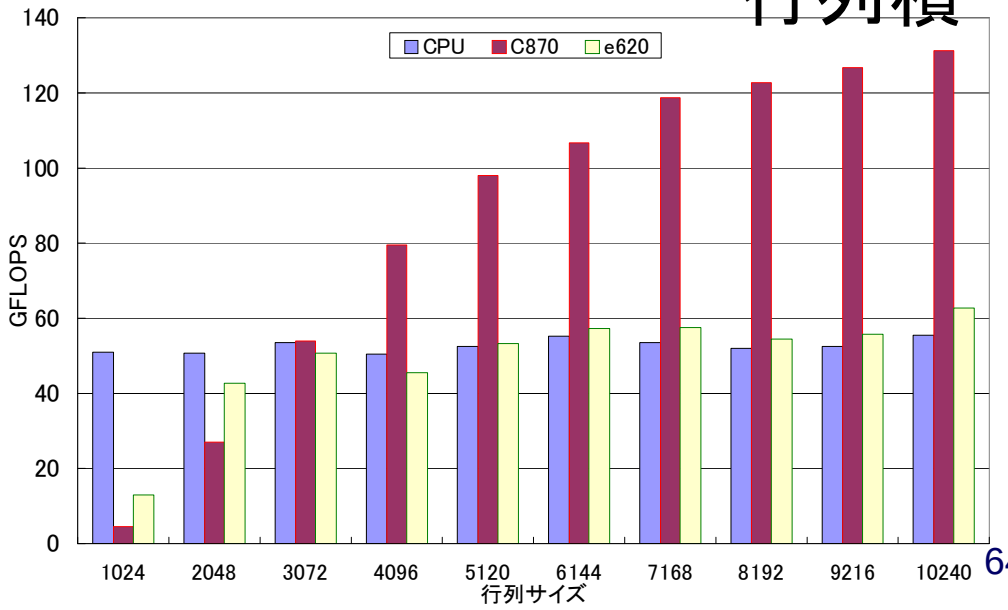
- 標準のライブラリで、FFTと行列積の性能を比較

- 行列積
  - 転送量が演算量に比べて、相対的に少ない
    - GPGPUに有利
- FFT
  - やはり、GPGPUは、演算が早い

FFT



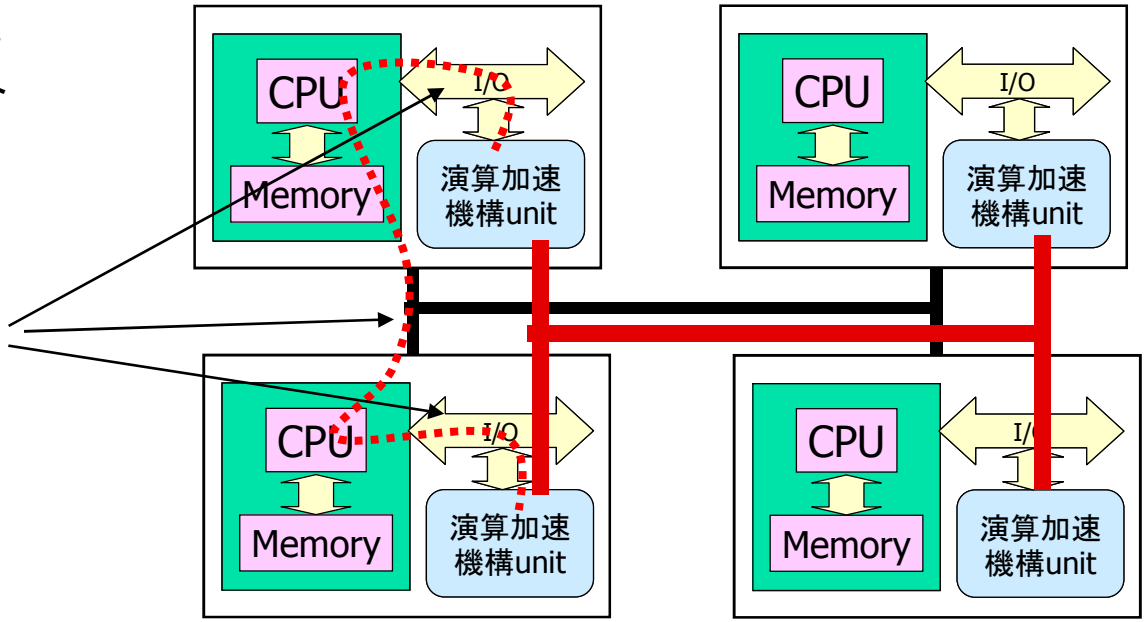
行列積



演算加速機構の問題点

- 演算加速機構(GPGPU)のメモリにデータがおいておける場合には、非常に効果的
  - 特にメモリバンド幅が大きい。グラフィックと融合した場合。
    - 姫野BMTで、メモリバンド幅効率80%超、理論ピーク142GB/sに対して、実効で115GB/sを実現 [成瀬2009]
- 1つのGPGPU以上に性能を上げようとおもうと、…
  - GPGPUのボードを複数装着
  - 演算加速機構を持ったノードで並列化
    - 現在のところ、余り有効ではない
    - IOの速度が非常に遅い
    - GPGPU同士の直接結合

ここがおそい！



おわりに―HPCとゲームの”深いぃ関係”（その3）

- ハイエンドのスパコンでも、これからの高性能化に向けて、Strong Scalingが重要になってきた。
  - ゲームで育ったGPGPUなど、演算加速機構が注目されている。
- Strong Scalingを可能とする、スパコン向けに開発・利用されるテクノロジーは、ゲームでも直接使える可能性が高い。⇒　目的が、共有できるようになってきた。
  - 次世代のGPGPUアーキテクチャ＝メニーコアプロセッサ（e.g. Intel Larrabee, 次のCell BE?)
  - 演算加速機構を結合するためのIOインタフェース(e.g. PCI Gen3)
  - 演算加速機構間のインターコネクト
- ゲームやPCに使われるテクノロジでないと、スパコンにも使えない。⇒　スパコンが生き残るための"エコシステム”問題
  - スパコンはそれ自体では投資を回収できない

ご静聴、ありがとうございました。