HPC用の高生産並列プログラミング言語

田浦健次朗

2011年5月27日 SACSIS

背景—並列計算機

- ノード内並列性が増加している
- ノードがヘテロ化している
- ノード数が増大している (Jaguar 20 万)
- 通信 (bytes)/計算 (flops) 比が**減少, 階層化**している
  - メモリ
  - ネットワーク
- マシン間の差異も増大している
- エネルギー効率などの設計基準が求められている
- ソフトによる耐故障が不可避になっている

$\Rightarrow$ プログラミングがますます困難に

潮流

高水準言語・高生産性言語への「関心」ないし「要求」が高まっている

- **プロジェクト:** DARPA High Productivity Computer Systems, DARPA Ubiquitous HPC, International Exascale Software Project
- **HPC 言語:** UPC, CAF, X10, Chapel, XcalableMP, ...
- **ノード内並列:** OpenMP, Intel TBB, ArBB, ...
- **DSL:** Liszt (mesh), Latin (data), Pregel (graph), ...

本チュートリアル

- 高生産言語とは? 何を目指しているか? 目指せばよいのか?
- 今日使える高生産言語をいくつか紹介
  - 共有メモリ: **Cilk**, TBB(少し)
  - 分散メモリ: **Chapel**, X10(少し), UPC(少し)
- 実際に使ってみて...

Disclaimer

- 高級言語は非常にたくさんあり, 研究されている
- 話者の能力と許された時間では, 全体のサーベイは不可能
- システムソフトウェアの研究では, **論文内容と実装のギャップ**もしばしば
- 今日の話は実際の処理系を使って確かめ・得られた知見に基づくものが中心
- 以下にコード, (東大情報基盤センター機関誌に連載中の) 解説記事などを公開している

  `http://www.logos.ic.i.u-tokyo.ac.jp/~tau/highly_productive_langs/`

高生産言語の審美眼

- **大域的な視点** ($\leftrightarrow$ **断片的な視点**) でのプログラミング
  - 「各プロセスの処理」ではなく，「処理全体」を記述
  - 「各プロセスのデータ」ではなく，「データ全体」を記述
- 制限のない並列性の記述 (**タスク並列**)
  - 多重ループの並列実行
  - 再帰呼び出しの並列実行, 任意文の非同期実行
- データの分割・配置と処理の記述が独立
  - **分散オブジェクト・分散配列**

負け組であり続けないために

- 「きれいなだけのコード」と，「速度を追求した結果」を同列に比べていてはいけない
- 特に，「局所性の高い並列実行」≈ 「計算/通信比の高い並列実行」ができる—もしくは書ける—ことは中心的課題かつ，**生きていくための最低条件**
- あらゆる階層 (コア間 → ソケット間 → ノード間 → スイッチ間) での局所性の維持をどう書かせるのか，という問題を解決する「枠組み」として設計していくのが重要であろう

例題

以下の一次元ステンシルコードの並列化を題材に, 前述の「審美眼」の中身を具体的に説明

```
for (i = 0; i < N; i++)
  a[i] = ... a[i+1] ...;
```

以下で言及する言語

- 共有メモリが前提: **OpenMP, Cilk, TBB (注)**
- 分散メモリをサポート: **MPI, UPC, Chapel, X10**

(注) TBB = Intel Threading Building Block

大域的/断片的な処理の記述

- **大域的な視点:** 「処理全体」を記述

```
forall (i = 0; i < N; i++)
  a[i] = ... a[i+1][j] ...;
```

+ 処理を「どう分割するか」の指示 (適宜)

- **断片的な視点:** 「個々のプロセス」の動作を記述 (MPI)

```
begin_i = ... MY_RANK ... N_PROCS ...
end_i   = ... MY_RANK ... N_PROCS ...
forall (i = begin_i; i < end_i; i++)
  a[i] = ... a[i+1] ...;
```

星取表

	処理全体を記述
OpenMP	Y
Cilk	Y
TBB	Y
MPI	
UPC	Y
Chapel	Y
X10	Y

大域的/断片的なデータの記述

- **大域的な視点:** 「データ全体」を記述

```
double a[N];
```

  + データを「どう分割するか」の指示 (適宜)

- **断片的な視点:** 「個々のプロセス」のデータを記述

```
double a[N / N_PROCS];
```

星取表

	処理全体を記述	データ全体を記述
OpenMP	Y	-
Cilk	Y	-
TBB	Y	-
MPI		
UPC	Y	Y
Chapel	Y	Y
X10	Y	Y

多重ループの並列実行

- 最も単純な例は以下のような多重ループ

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    a[i][j] = ... a[i+1][j] ...;
```

- OpenMP では「矩形
  の完全多重ループ (perfectly nested loop)」のみサポート

```
/* OK */
#pragma omp for collapse(2)
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    a[i][j] = ... a[i+1][j] ...;
```

- UPC も実質的には同等の制限

OpenMP の多重ループサポート (1)

- 容易に平坦化できる場合に制限
- 完全多重ループでないものは NG

```
/* NG */
#pragma omp for
for (i = 0; i < N; i++)
  s = ...;
  for (j = 0; j < N; j++)
    a[i][j] = ... a[i+1][j] ...;
```

OpenMP の多重ループサポート (2)

- 内側のループが別の関数という場合も NG

```
/* NG */
foo() {
#pragma omp for
  for (j = 0; j < N; j++)
    a[i][j] = ... a[i+1][j] ...;
}

main() {
#pragma omp for
for (i = 0; i < N; i++) foo();
}
```

再帰呼び出しの並列実行

- 分割統治法
- 組み合わせ探索

などで用いられる

- Cilk:

```
void quicksort(a, l, r) {
  ...
  spawn quicksort(a, l, p - 1);
  spawn quicksort(a, p, r);
  sync;
}
```

任意の文の非同期な実行

- 再帰呼び出しの並列化をサポートしていれば, 仕組みとしてはほぼ同じ
- Chapel:

```
void quicksort(a, l, r) {
  ...
  cobegin {
    quicksort(a, l, p - 1);
    quicksort(a, p, r);
  }
}
```

タスク並列のサポート

- 単に文法として許せばいいというものではなく, **実装は単一の for ループと大きく異なる**
- **すべての並列度が「一斉に生まれる」か否か**が分かれ目 (入れ子でない for ループや, 完全多重ループでは YES)
- YES の場合, 負荷分散 (実行可能タスクの管理) が簡単
  - 固定数のワーカに静的に分割 (OpenMP の static)
  - カウンタで動的分割 (同 dynamic, guided) など
- 詳しくは Cilk の節で

星取表

	処理全体を記述	データ全体を記述	入れ子/再帰タスク並列
OpenMP	Y	-	△
Cilk	Y	-	Y
TBB	Y	-	Y
MPI			
UPC	Y	Y	
Chapel	Y	Y	△
X10	Y	Y	Y-

構文としてサポートしていることと, 実装がよいかどうかは別の話

データの分割・配置と処理の記述が独立

計算・データの配置に, 処理の記述が左右されない

- UPC:

```
upc_forall (i = 0; i < N; i++; continue)
  upc_forall (j = 0; j < N; j++; ...)
    a[i][j] = ... a[i+1][j] ...;
```

- MPI では...

```
for (通信相手) {
  MPI_Irecv(...);
  送信データの集約;
  MPI_Isend(...);
}
begin_i = ...
end_i = ...
begin_j = ...
end_j = ...
for (i = 0; i < end_i - begin_i; i++)
  for (j = 0; j < end_j - begin_j; j++)
    a[i][j] = ... a[i+1][j] ...;
```

データの分割・配置と処理の記述が独立

- HW 共有メモリが前提であれば, 当たり前
- SW でも, そのような記述を許すだけなら大きな困難はない
- 性能上の課題が多い
  - 局所性の制御の記述
  - ローカルデータへのアクセスオーバーヘッド
  - 非連続な遠隔データへのアクセスの集約
- 後に Chapel, X10 の設計について述べる

星取表

	処理全体を記述	データ全体を記述	入れ子/再帰タスク並列	データと処理の独立
OpenMP	Y	-	△	-
Cilk	Y	-	Y	-
TBB	Y	-	Y	-
MPI				
UPC	Y	Y		Y
Chapel	Y	Y	△	Y
X10	Y	Y	Y-	△

実例紹介

- **Cilk**, TBB(少し)
- **Chapel**, X10(少し), UPC(少し)

Cilk : ハイライト

- Cilk (MIT) → Cilk++ (Cilk Arts) → Cilk plus (Intel)
  - Cilk 5.4.6 http://supertech.csail.mit.edu/cilk/
  - Cilk plus : Intel parallel composer などの一部. Intel compiler のみサポート
- spawn, sync という少ない構文 (タスク並列のみ) で並列化
- Cilk plus には並列 for 文などのサポートがある
- **ポイント: Work stealing による動的負荷分散**

spawn と sync

- spawn：

  spawn 関数呼び出し;

  で関数呼び出しを非同期に実行
- sync：

  sync;

  で, その関数がこれまで行った spawn すべての終了を待つ
- 基本はこれだけ

Cilkでの「ループの並列化」

- 単純なループ

```
for (i = 0; i < N; i++)  S( i);
```

も,

- Cilk 流では分割統治

```
cilk void rec(int l, int h) { /* [l, h) を実行 */
  if (h - l == 1) { S( l); }
  else {
    spawn rec(l, (l + h) / 2);
    spawn rec((l + h) / 2, h);
    sync;
  }
}
```

多重ループも同様

```
for (i = 0; i < N; i++)
  for (i = 0; i < N; i++)
     S( i,  j);
```

の並列化

多重ループの並列化

```
cilk void rec(int l0, int h0, int l1, int h1) {
  /* [l0, h0) x [l1, h1) を実行 */
  if (h0 - l0 == 1 && h1 - l1 == 1) { S( l0, l1); }
  else if (h0 - l0 >= h1 - l1) {
    /* l0, h0 の真ん中で分割 */
    spawn rec(l0, (l0 + h0) / 2, l1, h1);
    spawn rec((l0 + h0) / 2, h0, l1, h1);
  } else {
    /* l1, h1 の真ん中で分割 */
    spawn rec(l0, h0, l1, (l1 + h1)/2);
    spawn rec(l0, h0, (l1 + h1)/2, h1);
  }
}
```

注

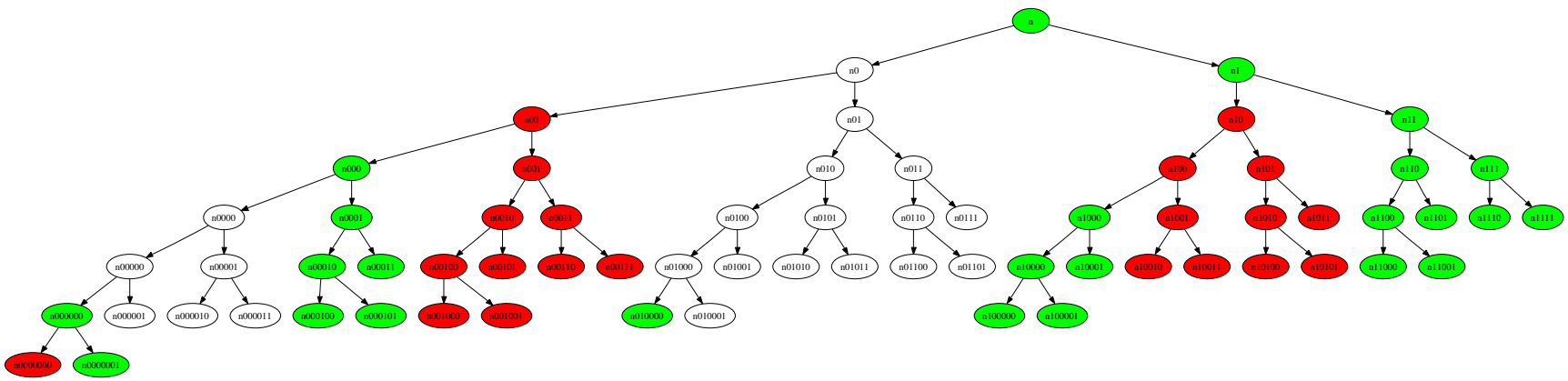
- もちろん区間長 1 になるまで再帰呼び出しを深くする必要はない. 適当なしきい値で逐次ループに切り替える
- 「プロセッサ数程度のリーフができるように」, しきい値を調整する必要はなく, それで得もしない. 理由は Cilk の実装方式にある
- 多重ループでは, 多次元の矩形を表す構造体を用いれば, よりすっきり書けるし, 実際 **TBB, Chapel, X10 などでサポート**されている

面倒なだけでは?

- もちろん, 上記に対する syntax sugar としての並列 for 文は実用上は重要 (で, 実際 Cilk plus では提供されている)
- ポイント
  1. **均整のとれた直方体への分割を自然に記述できる**こと
  2. どこまで深く分割するか, 利用可能なプロセッサ数に応じて**適応できる**こと

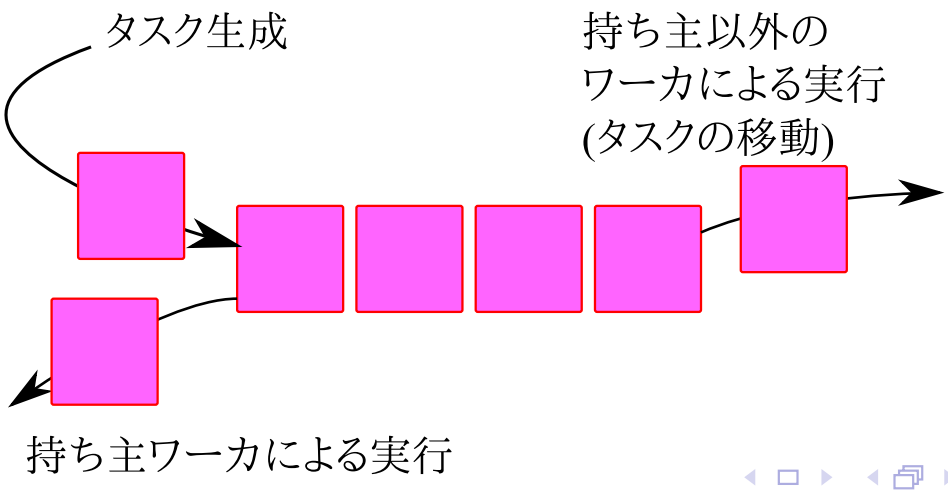
Cilk の負荷分散方式 (1)

- ルーツは並列 Lisp の Lazy Task Creation
- 再帰呼び出しで作られた木構造を，「なるべく大きな固まりで」負荷分散



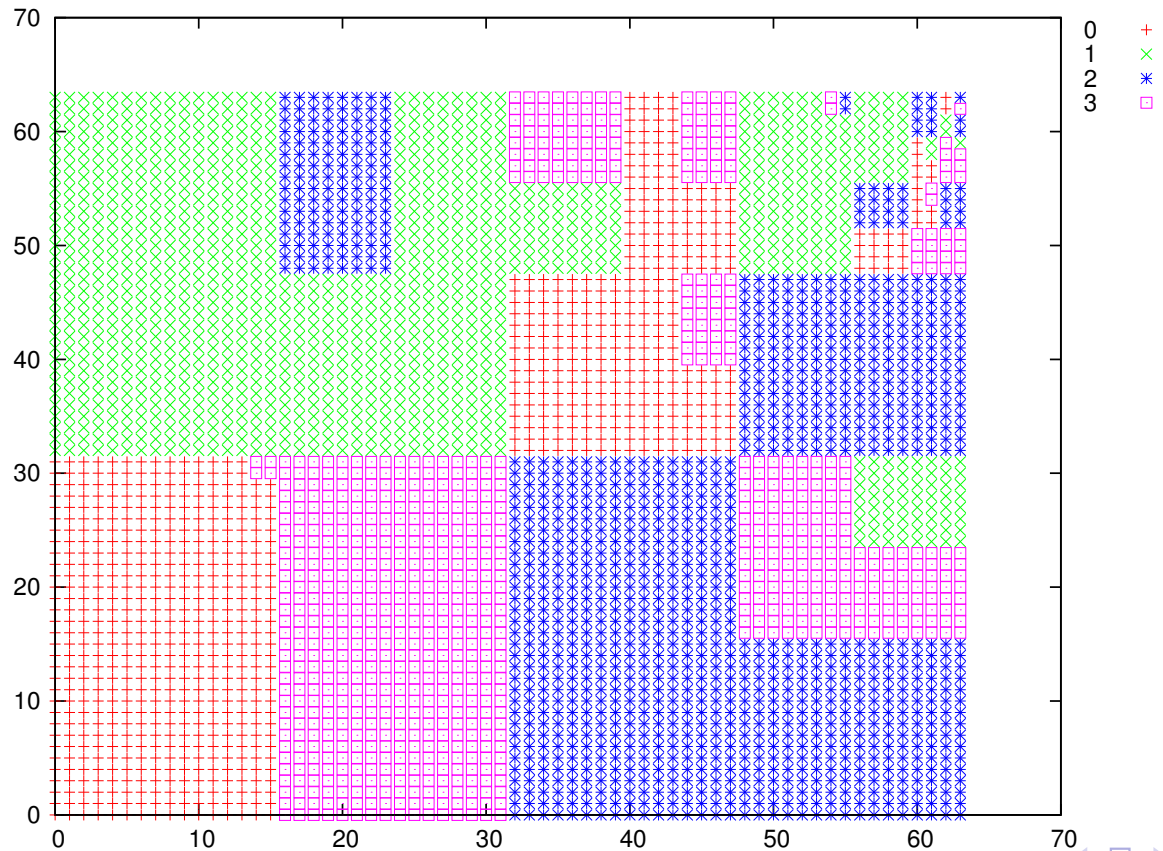
Cilk の負荷分散方式 (2)

- 各ワーカが一つタスクキュー (deque) を持つ
- **タスク生成時:** タスクを PUSH; すぐにそのタスクを実行 (≈ 関数呼び出し)
- **タスク終了時:** タスクを POP; 親に戻る (≈ 関数からのリターン)
- **暇なとき:** 他のワーカから盗む. ただし, **PUSH/POP と逆の側 (木の根に近い側) から.**



負荷分散の実例

- 2 次元のループ (64×64) を直方体分割で分割した例
- 4 ワーカで実行. 実行したワーカで色分け



注目に値する理由

- 動的負荷分散の余地を残す
- マシン独立に，**「計算/通信比」の高い分割を「プログラマが」記述できる**

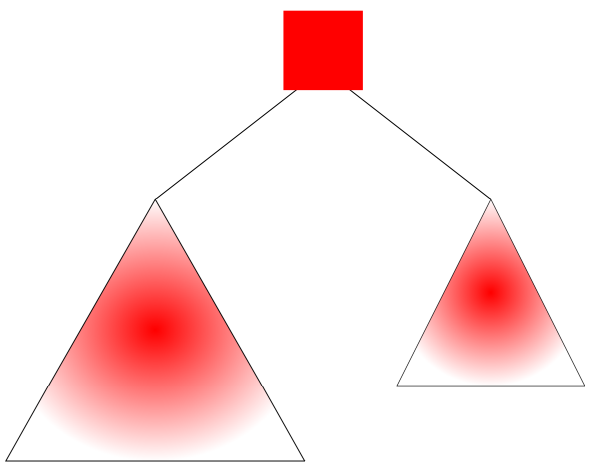
Cilk (LTC) の実行方式は，「逐次で連続して実行されるコード」を，1 ワーカ上で，それも連続して実行する傾向にある
⇒ もともと逐次コードにあった局所性をよく保存する

ポータブルかつ計算/通信比の高い分割の原則

プログラマに対する要請:

- 問題全体を, 「どのプロセッサにこの部分」というレベルまで細かく分割させる代わりに,
- 2 つ (ないし 3, 4, ...) に分けよ. ただしその際,
  - **部分問題 (再帰呼び出し) ごとの計算/通信費を保ち**,
  - **部分問題のアクセスする領域が減る**,

  ように分割する



密行列積への適用

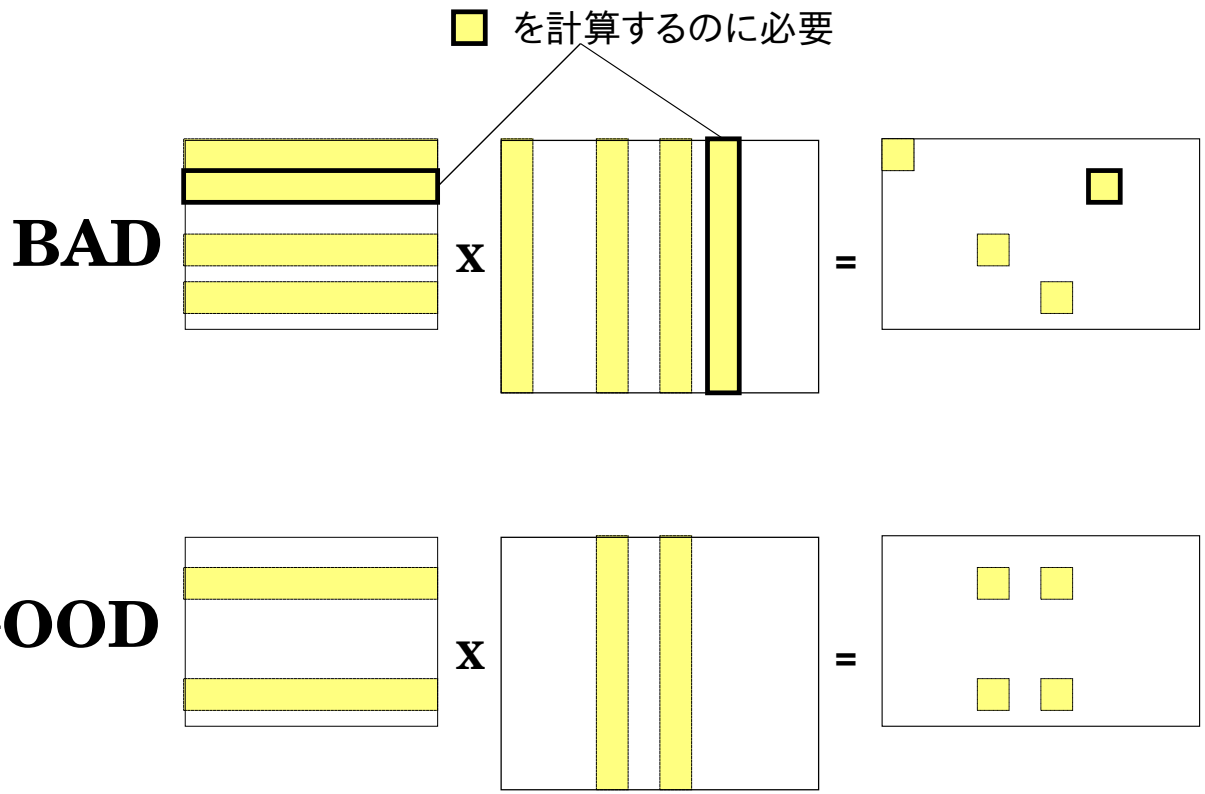
考え方:

$$A \times B = C$$

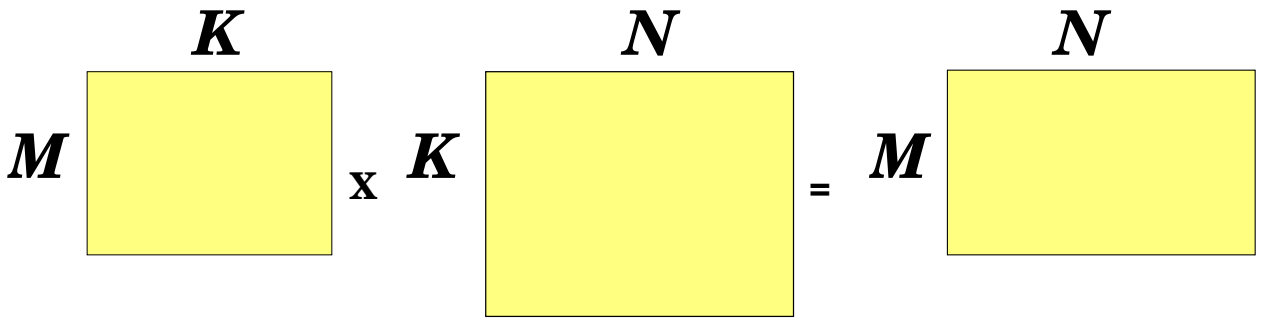
における計算の「一部分」(e.g., 半分) を担当する際, どの一部分を担当するのが計算/通信費が高くなるか

1. (明らかに), 同じ計算量 (C の中の面積) ならば「寄せ集めれば長方形」という領域を計算するのが良い. バラバラはダメ
2. 同じ長方形でも, 正方形に近いほうが計算/通信費がよい
3. 横長 $\times$ 縦長 にしてはいけない (分割してもアクセス領域が減らない)

バラバラはダメ



C の同じ面積を計算するなら正方形に近く



$$\frac{\text{計算量}}{\text{データアクセス量}} = \frac{2MNK}{MK + NK + MN}$$

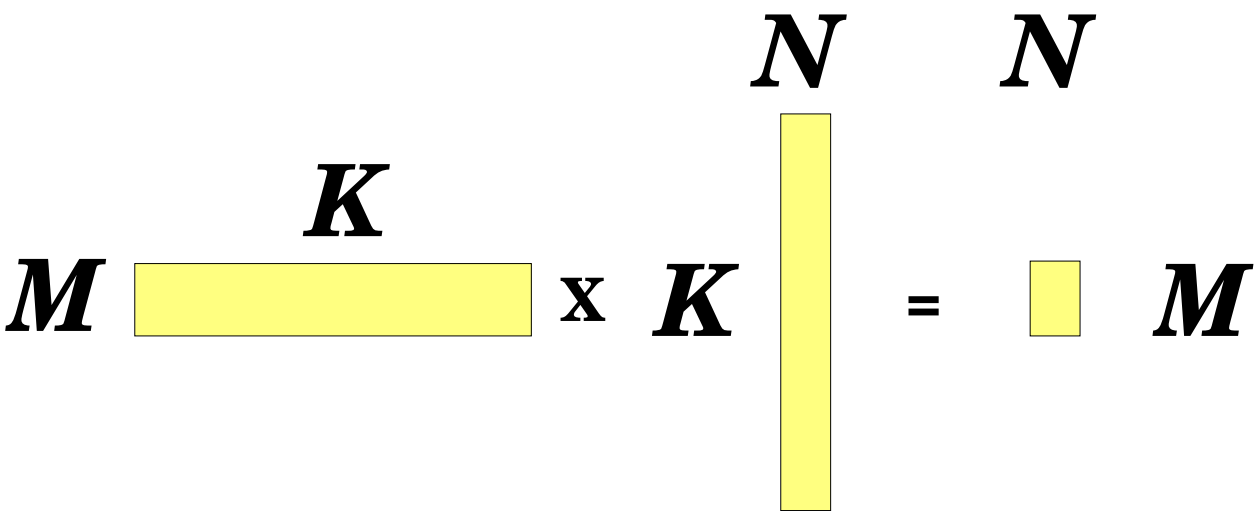
- M が大きければ...$\Rightarrow$ C を縦に割る

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

- N が大きければ...$\Rightarrow$ C を横に割る

$$A(B_1 \quad B_2) = (AB_1 \quad AB_2)$$

横長 × 縦長 = 小さいの は避ける



- M も N も (K に比べて) 小さければ... アクセス量が減らないのでどちらを割っても良くない

$$(A_1 \;\; A_2) \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2$$

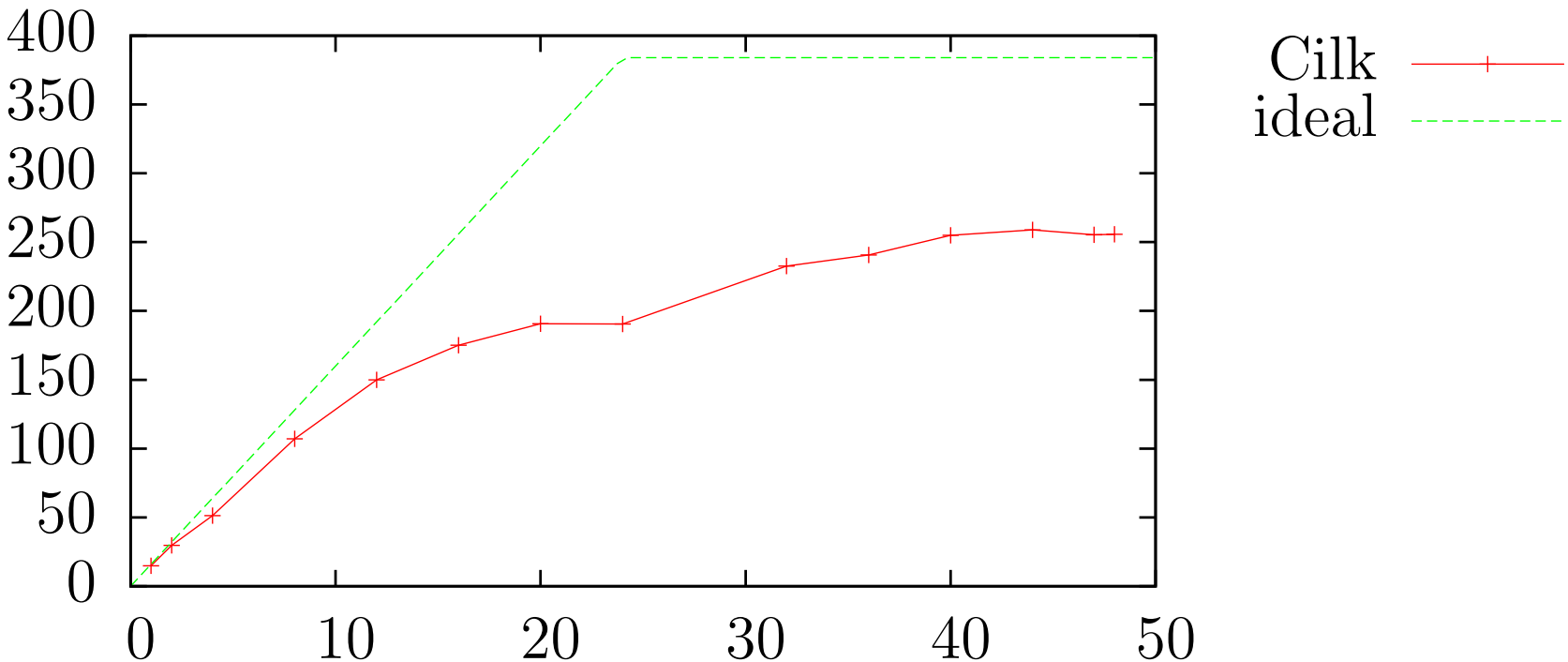
と分割

Cilkでの行列積 (Cache-Oblivious Algorithm)

```
/* A : m x k,  B : k x n,  C : m x n, 行優先. 列数 ld. */
cilk void rec_matmul(REAL *A, REAL *B, REAL *C,
                     int m, int k, int n, int ld, int add) {
  if ((m + n + p) <= しきい値) {
    普通に計算;
  } else if (max(m, k, n) == m) {  /* 縦に割る */
    spawn ...; spawn ...; sync;
  } else if (max(m, k, n) == n) {  /* 横に割る */
    spawn ...; spawn ...; sync;
  } else {   /* (A1 A2) * |B1| = A1 B1 + A2 B2
                          |B2|                */
    spawn ... ; sync ; spawn ...; sync ;
  }
}
```

Cilk 行列積の性能

- Xeon E7540 2.00GHz, 24 物理コア 48 スレッド
- 入力: 16384 × 16384 float
- 256×256 以下は GotoBLAS で (32768 タスク)



SMP, NUMA, 分散メモリ, . . . (1)

- もちろん Cilk は HW 共有メモリが前提
- 一方「通信/計算比を高く保つ」「部分問題のデータアクセスを減らす」分割の考え方は, 分散メモリであろうと常に有効
  - **メインメモリ =L4 キャッシュというアナロジー**
  - 通信性能の違いも, 究極的にはパラメータの違いに過ぎない
- では, Lazy Task Creation は分散メモリでもうまくいくということか?
- ある程度は yes (cf. Ibis/Satin)
- だが直感的には HPC 向けには crazy にも見える. . .
- パラメータ以外に何が違うのか?

SMP, NUMA, 分散メモリ, . . . (2)

- 分散メモリと共有メモリは, パラメータ以外に何が違うのか?
  - 共有メモリ: データが (L3) キャッシュに収まらないことが前提 ⇒ どこで計算をしてもデータをなめる分くらいの L3 ミスは前提
  - 分散メモリ: データはどこかのノードのメインメモリ (L4 キャッシュ) にある ⇒ そこで計算することで L4 ミスはほとんどなくせる. 分散メモリでの通信削減というと普通はこのレベルの話をする
- 適切な拡張で, あらゆる階層の局所性を高める統一的な考え方に発展できる可能性があると思います

優れた分割の簡便な表現は鍵

- 行列積や ORB などの分割方法を美しく (≈ 大量の区間パラメータを引き回さずに) 記述するには, 「多次元の矩形領域」のサポートがあると有効
- TBB : **block_range2d, block_range3d**
- Chapel : **domain**
- X10 : **Region**

TBB (in 1 スライド)

- C++のライブラリ (言語拡張は無し)
- 並列 for は parallel_for という C++の「関数」に, イテレーション空間と, その一部を実行するオブジェクトを渡す

```
parallel_for(
  blocked_range3d<int>(a0,b0,a1,b1,a2,b2),
  do_iterations(...));
```

- 入れ子やタスク並列もサポート

Chapel : ハイライト

- DARPA HPCS プロジェクトに対する Cray の提案
- 謳い文句: Productivity = Performance + Programmability + Portability + Robustness
- ポイント:
  - **データのビュー:** 大域アドレス空間. オブジェクトや配列は分散透明
  - **計算のビュー:** 大域的視点で記述. タスク並列・データ並列などリッチな構文
  - **領域 (domain):** 一級の「domain データ型」で多次元矩形領域や不規則領域などを統一的に表現
  - **ノードへのデータや計算の配置:** プログラマが制御 (可能) + デフォルトルール (owner computes など)
- 主観: 学んで気分がいい言語

Domain データ型 (1)

- 一次元の区間 $[a, b]$, 多次元の区間 (矩形) $[a, b] \times [c, d]$, などを簡便な記述でサポート

```
var my_range = a..b;          // [a,b] 区間
var my_1d_dom = [c..d];       // [c,d] 1D 領域
var my_2d_dom = [0..3,4..6];// [0,3]x[4,6] 2D 領域
```

領域は, 1 級のデータ (変数に入れたりご自由に).

- 配列は, domain から値への写像.

```
var a : [9..10] int;          //  [9,10]  -> int
var b : [my_2d_dom] real; //  [0,3]x[4,6]  -> real
```

Domain データ型 (2)

- domain は第一義的には配列の添字に用いるが, 単に「多次元の繰り返し空間」を表すために用いるのも有効
- 例 1:

```
// 2 重ループに相当
for (i,j) in [0..9,0..9] { ... }
```

- 例 2 (ORB 相当):

```
proc orb(space : domain(2)) {
  if (small(space)) { ... }
  else {
    var (s1,s2) = bisect(space);
    orb(s1); orb(s2);
  } }
```

並列構文 — タスク並列 (1)

- **begin** 文

```
begin 文
```

で任意の文をタスクとして (非同期に, ローカルで) 実行.

- **sync, single** 変数. 他のタスクと同期を取りたければ, sync または single 変数を通じて.

```
var x : single int;
begin x = fib(n - 1);
var y = fib(n - 2);
return x + y;
```

single は単一代入, sync は容量 1 有限バッファ

並列構文 — タスク並列 (2)

いくつかの糖衣構文:

- **cobegin**：いくつかを非同期に実行して全部の終了待ち

```
cobegin { 文 文 ... }
```

- **coforall**：ループの各繰り返しを非同期に実行して全部の終了待ち (cf. データ並列用の forall)

```
coforall 変数 in イテレータ do 文
coforall 変数 in イテレータ { 文 ... }
```

イテレータ

- Chapel では for 文およびその仲間は, イテレータをとって繰り返しを実行する
- 自分で作ることもできる. さしあたりここでは領域や配列がイテレータになるとだけ理解しておけばよい
- 遅そう? ⇒ イテレータが単純な区間の場合, 完全に unbox 化されたループにコンパイルされる

```
for i in d foo(i);   ===>
```

```
while (T5) {
  foo(i);
  T7 = (i + 1);
  i = T7;
  T8 = (T7 != end);
  T5 = T8;
}
```

並列構文 — データ並列 (1)

- **forall** 文

```
forall 変数 in イテレータ do 文
forall 変数 in イテレータ { 文 ... }
```

- coforall と何が違う? ⇒ forall は処理系が逐次化するかもしれない (例えば, i 番目の繰り返しが $(i+1)$ 番目を待つような同期は, coforall でのみ合法)

並列構文 — データ並列 (2)

これであなたもきっと好きになる!?

- **forall** 式 (ほとんど関数型!)

```
forall 変数 in イテレータ do 式
```

これ自身がまたイテレータ

- **reduce/scan** 式

```
+ reduce イテレータ
+ scan イテレータ
```

Locale — 分散メモリノードの抽象化

- Locale オブジェクト ≈ 計算に参加しているノードひとつ
- **Locales**：計算に参加しているノードの配列が格納された大域変数
- 各計算がどこで行われるかは明確に規則化されている
  - main 関数が `Locales[0]` で実行
  - 基本はローカルで継続
  - **on 構文**で任意の文・式の実行場所を指定
  - 分散イテレータ (後述) は owner computes

Locale と on 構文

- on 構文:

```
on (locale 式) do 文
on (locale 式) { 文 文 ... }
```

- 例:

```
var a : [0..3] string;
on (Locales[3]) { a[1] = my_hostname(); }
writeln(a[1]); // Locale[3] のホスト名
```

- 例 2: 任意のデータにはその居場所がある

```
/* a[1] の居場所で実行 */
on (a[1]) { a[1] = my_hostname(); }
```

分散配列 — コンセプト

- Domain から locale への写像を作ることで, mapped domain が作られる
- Mapped domain を domain (添字集合) とした配列が分散配列
- いわゆるブロック分散

```
use BlockDist;
// 普通の domain
var l_dom : domain(2) = [1..n,1..n];
// 分散 domain
var d_dom : domain(2) dmapped new dmap(
    new Block([1..n,1..n])) = [1..n,1..n];
var la : [l_dom] real; // 普通の配列
var a  : [d_dom] real;  // 分散配列
```

Mapped domain を用いた並列 for

```
var d_dom : domain(2) dmapped ...
var a : [d_dom] real;
var b : [d_dom] real;
forall (i,j) in d_dom {
  b[i,j] = a[i+1,j] + a[i,j+1] + a[i-1,j] + a[i,j-1];
}
```

これで,

- ノード間並列実行
- イテレーション (i, j) は a[i, j], b[i, j] の存在する locale で実行
- ノード内並列実行

が表現されている

Chapel : まとめ

- 「直交性」の高い・美しく・うなづける設計
  - domain (1 次元・多次元・スパース)
  - タスク並列構文と on
  - 普通の domain/配列と分散 domain/配列
- ノード間のデータ・タスク配置については「現実的選択」

X10 : ハイライト

- Chapel と X10 は「非常に」よく似ている
- 名前が違うだけ, という概念が至るところにある

	Chapel	X10
タスク並列構文	begin	async
タスクの待ち合わせ	sync { . . . }	finish { . . . }
並列ループ	forall	N/A
ノードの抽象化	Locale	Place
ノード間移動	on	at
多次元配列の添字	タプル	Point
一次元の区間	range	Region
多次元の矩形	domain	Region
不規則な領域	domain	N/A
遠隔参照	透明	明示的

X10 : 特徴的な点

- データは分散透明ではない
- データは基本的にはローカル
- at 構文による計算移動時のデータの移動
  - val (単一代入変数) の中身 ⇒ コピー
  - var (更新可能変数) の中身 ⇒ 参照禁止

```
val a = new Array[String](10, "hello");
var b : String = "wao";
at(p) {
  a(0) = "hage"; // コピーを更新
  a(1) = b;      // 禁止: (コンパイル) エラー
}
```

X10 における遠隔参照

- データが基本的にローカルならば, 遠隔参照 (ノード間でのデータ共有) はどうやるのか?
- 明示的な遠隔参照 (**GlobalRef**) を作る
- 以下で, 配列 a への「参照」が place p に渡る

```
val a = new Array[Int](0..9);
val ga = new GlobalRef[Array[Int](1)] (a);
on (p) { .. ga .. }
```

再び...分散透明ではない

- GlobalRef の dereference (本体の取り出し) は,

```
val b = ga(); // 遠隔参照の dereference
```

- ただしこれは本体の存在するノードでのみ可能

```
val a = new Array[Int](0..9);
val ga = new GlobalRef[Array[Int](1)] (a);
on (p) { .. ga() .. } /* NG */
```

はエラー (2.1.2 では, C++
コードのコンパイルエラー. 意図かバグか?). 以下はOK.

```
on (ga.home) { .. ga() .. }
```

UPC in 1スライド

- 大域 (shared) 変数および配列. それらへのポインタ (shared ポインタ)
- shared 配列は 1 ノード集中もしくは block cyclic 分散
- local ポインタと shared ポインタを静的な型で明確な区別 (小オーバーヘッド重視)
- SPMD + OpenMP に似た, worksharing for (upc_forall)
- 各イテレーションを実行するプロセス番号は明示的に指示 (affinity 指示)

処理系の性能評価: プログラム

- Ising モデルのモンテカルロシミュレーション
- 1 タスク: 乱数を引きながら 2 次元配列を書き換える, ほぼ通信なし, 0.2 秒ほどの計算.

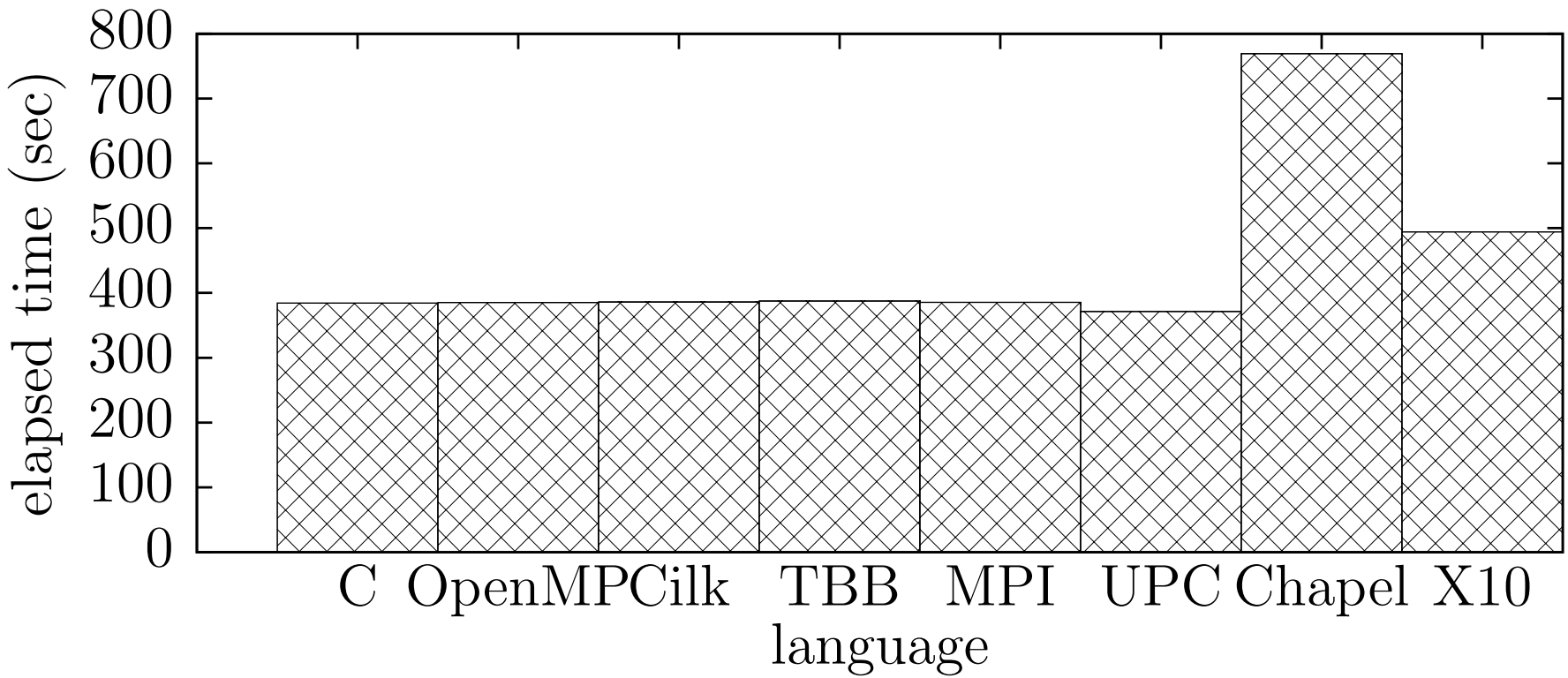
以下を 1024 回:

```
a[64][64]; /* 1 or -1 */
for (4096000 回) {
  (i,j) = 乱数 in [0,63] x [0,63];
  dE=a[i][j] が-a[i][j] に変化した時のエネルギー変化;
  if (乱数 in [0.0,1.0] < exp(-dE/T)) {
    a[i][j] = -a[i][j];
  } }
```

評価環境

- 東大 HA8000
- AMD 8356 2.3GHz 16 コア, 16 ノード
- GCC 4.5.2 (OpenMP)
- Cilk : 5.4.6
- TBB : 3.0 2010/9/15 (tbb30_20100915oss_lin.tgz)
- MPI : mpich, MX
- UPC : Berkeley UPC 2.12.1, gasnet on MPI
- Chapel : 1.3.0, gasnet on MPI
- X10 : 2.1.2 on mpich2

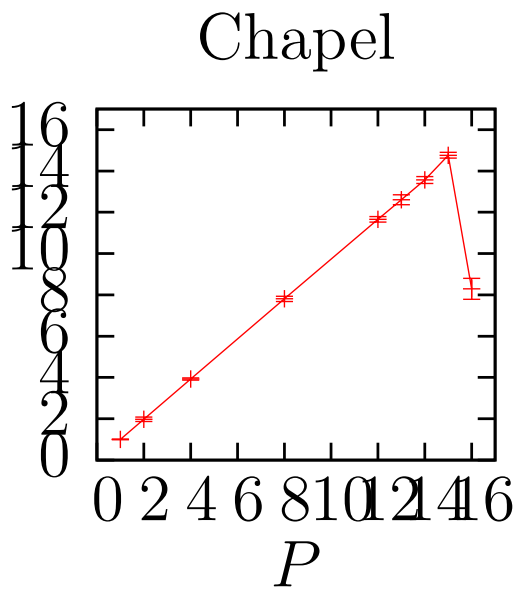
逐次性能比較

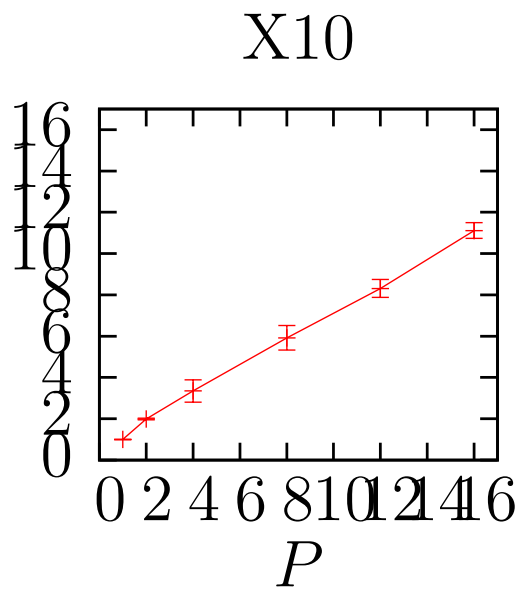

elapsed time (sec)
800
700
600
500
400
300
200
100
0
C
OpenMP
Cilk
TBB
MPI
UPC
Chapel
X10
language

逐次性能

- X10： 25%程度のオーバーヘッド
- Chapel： 100%程度のオーバーヘッド
- どちらも, 最適化オプションで性能が 5 倍以上向上

1ノード台数効果



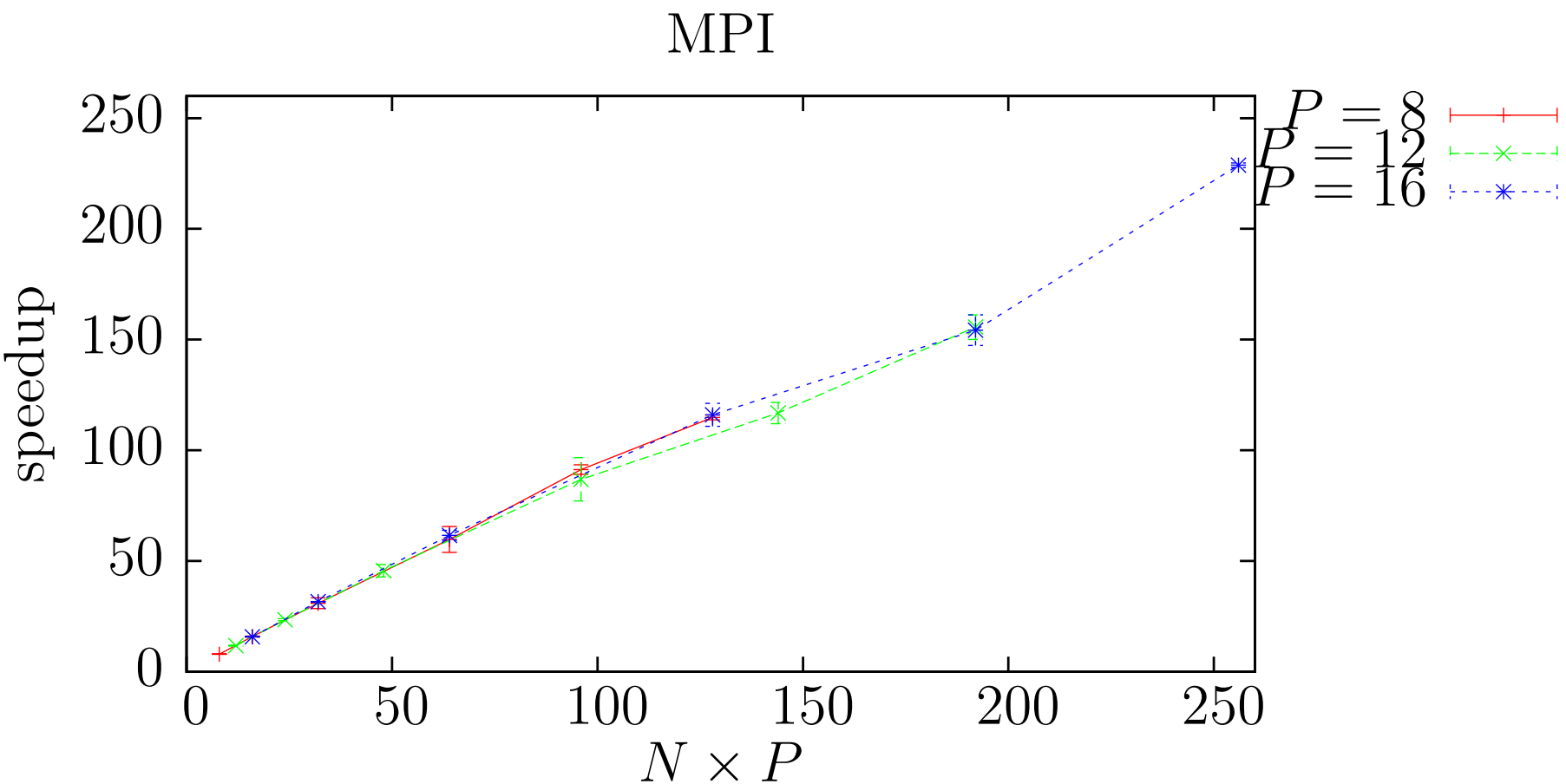


他の言語はほぼ 16 コアで 16 倍

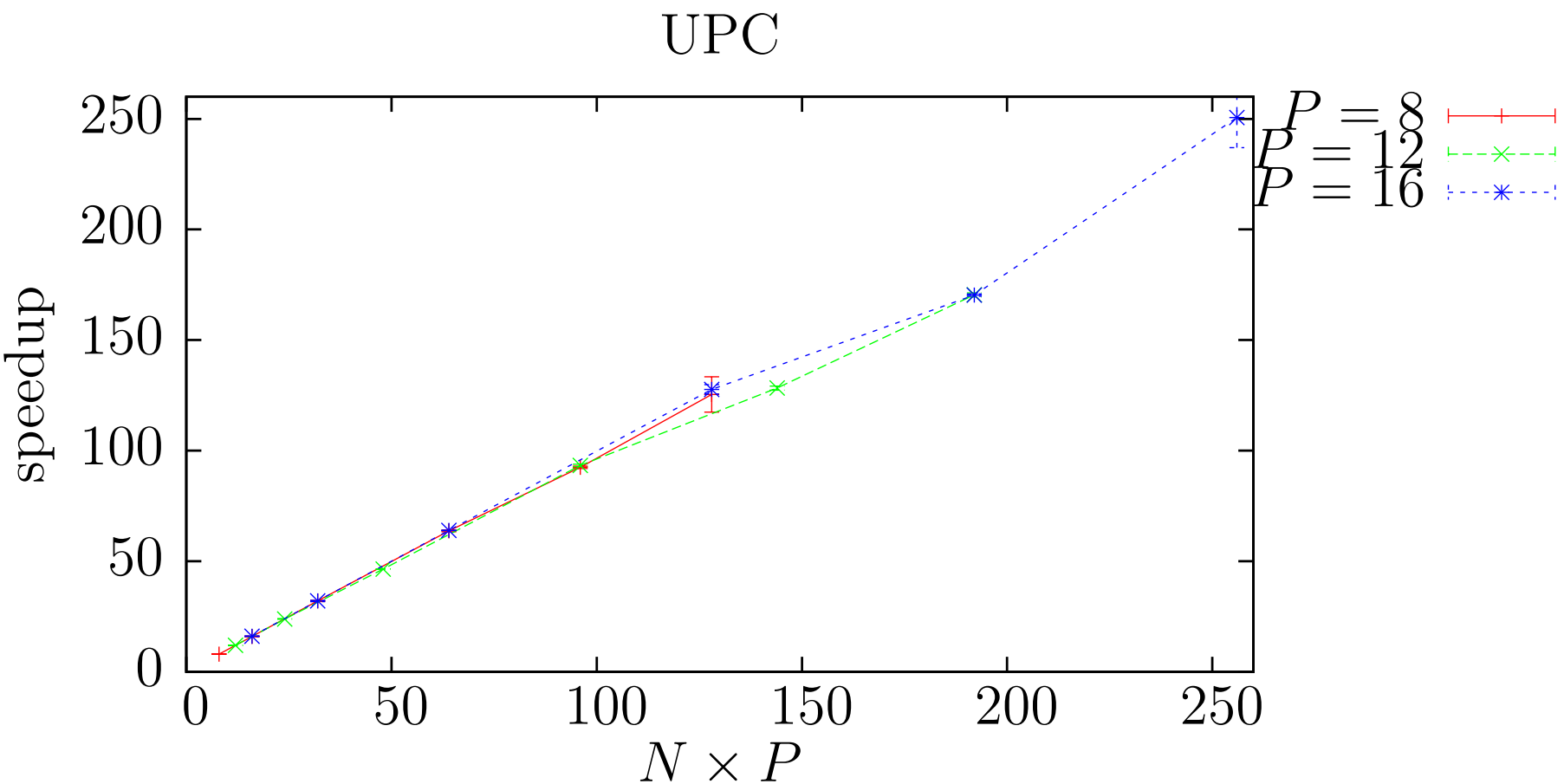
1ノード台数効果

- Chapel (ノード内はデータ並列を利用): 常に1コア分が余分に使われる. コア数分の並列度を出すと性能が悪くなる.
- X10 (ノード内もタスク並列 (async) を利用): 70%程度の効率

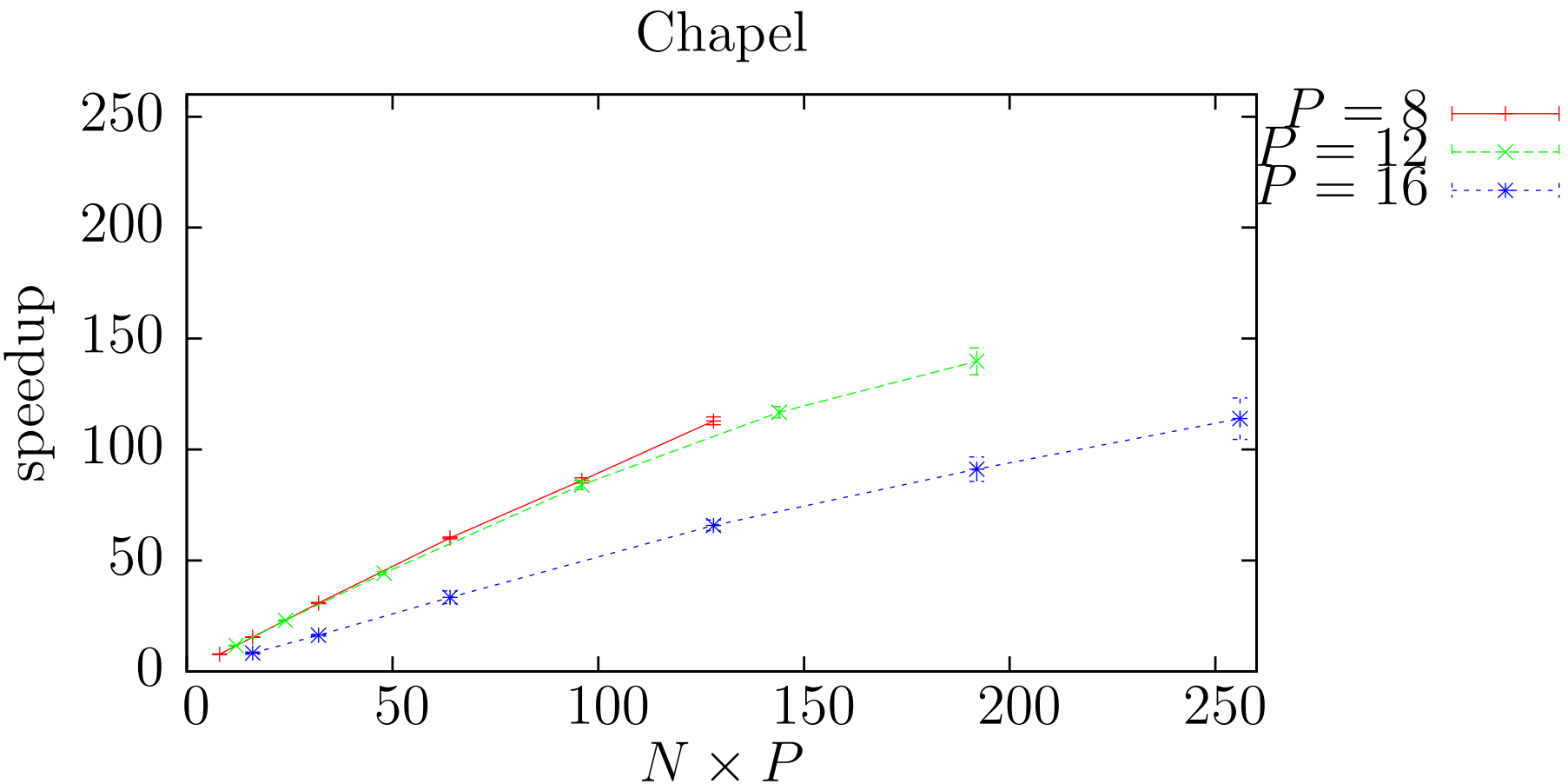
台数効果 (MPI)



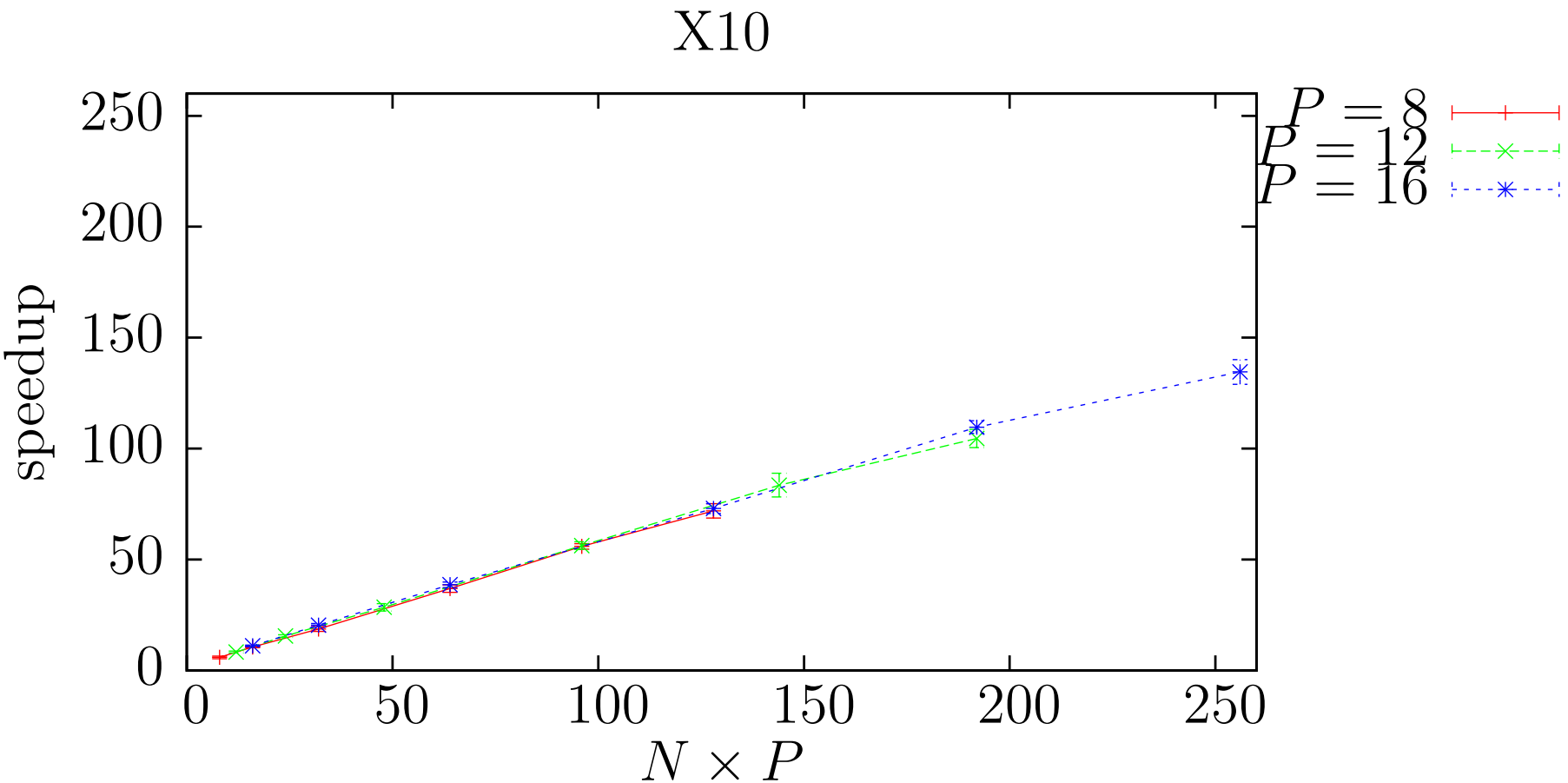
台数効果 (UPC)



台数効果 (Chapel)



台数効果 (X10)



まとめ

- 高水準言語は進化している
  - **ノード内:** 分割統治 + 動的負荷分散
  - **ノード間:** 多次元 domain の抽象化, データ分散の抽象化, domain の分割 (ORB, etc.), データ並列, ... 「ローカリティのよいコード」を書くための簡便な道具
- 「通信計算費を保存・アクセス領域を小さくする分割統治」はマシン非依存に, 局所性を保て, 動的に負荷分散できる有望パラダイム (だろう)
- 現状の分散メモリ用言語は, ノード間のデータ・計算のマッピングはすべて明示的

展望 (1)

- 通信の集約化と遅延隠蔽
  - データ並列構文でのデータアクセスの集約,
  - 並べかえ (遠隔アクセスするものを先に)
- ローカリティチェックオーバーヘッド
- 良いタスク並列の実装
- 分散メモリ管理 (GC)

展望 (2)

- マシンの詳細に非依存 かつ 実用的な性能理解が可能な性能モデル
  - **計算/通信費を元にした性能モデル**：プログラマへ露出
  - Lx キャッシュサイズ, ラインサイズ, スレッド, コア, アクセレレータ, ノード, ネットワークトポロジー, . . . : 知らなくても十分
- そのモデルを具現化した, マシン非依存な高水準言語