実践的シミュレーションソフトウェアの開発演習 (HPC 基礎)

平野 敏行

2016/04/14

はじめに

目的

- HPC プログラミングに必要な基礎を身につける
  - HPC ハードウェアの基礎知識
  - 並列プログラミングの基礎知識
  - テスト (基礎演習)

課題 基礎演習

目的

- Linux システム・MPI/OpenMP の使い方に慣れる
  - ファイル・ディレクトリの操作
  - テキストファイルの作り方・表示
- C/C++によるプログラミングを習得
  - ターミナルへの出力方法 (printf() etc.) の習得
  - バイナリファイルの読み書き (fopen(), get() etc.) を習得
  - 動的なメモリ確保・開放の方法を取得
  - コンパイル・実行の仕方
  - Makefile の書き方
- 並列処理
  - 簡単な MPI / OpenMP の並列計算の書き方・挙動を習得
  - 応用演習に備える

課題

- 以下を満たすプログラムを作成しなさい。
  - バイナリファイルで与えられた行列 A, B の積 C を計算する。
  - 行列 C を指定されたフォーマットでファイルに出力する。
- 最新情報・ヒントは wiki を参照すること
  - https://bitbucket.org/fumitoshi_sato/2016lecture/wiki/基礎演習課題 (行列積) について

注意事項

- 行列の次元はファイルに記録されている
  - (コードに決め打ちしないこと)
- 倍精度で計算・出力すること
- 並列計算すること
  - 短い時間で処理できること
  - 高い並列化効率を達成すること
  - BLAS などの行列演算ライブラリを使用しないこと
    - テストに使用することは可
    - サンプルは用意してあります
- 締切: 2016/05/21(木) まで
  - スケーラビリティのテスト (excel ファイル) も添付のこと

行列ファイルの仕様

- 先頭から 32bit 符号付き整数 (int) で行数、列数が順に格納される
- 倍精度浮動小数点型 (double) で (0, 0), (1, 0), (2, 0), ... (N-1, 0), (1, 0), ..., (N-1, N-1) の順に値が格納される
- FX10 を利用する場合、エンディアン (バイトオーダー) に注意すること!
  - ログインノードはリトルエンディアン
  - 計算ノードはビッグエンディアン

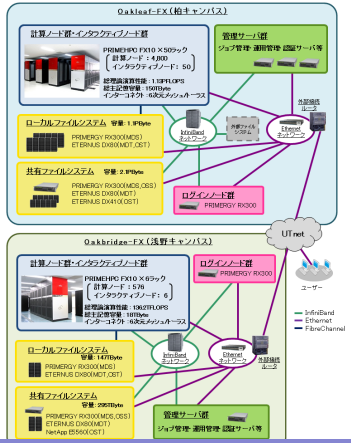
HPC 概略

スーパーコンピュータ

- 当時の最新技術が搭載された最高性能のコンピュータ
  - High Performance Computing (高性能計算)
  - 基本構成 (CPU, メモリ, ディスク, OS 等) はパーソナルコンピュータと同じ
  - 非常に高価
  - 最近の流行は分散並列型

FX10 システム概略

- http://www.cc.u-tokyo.ac.jp/system/fx10/fx10_intro.html



Top500 (http://top500.org/) (1/2)

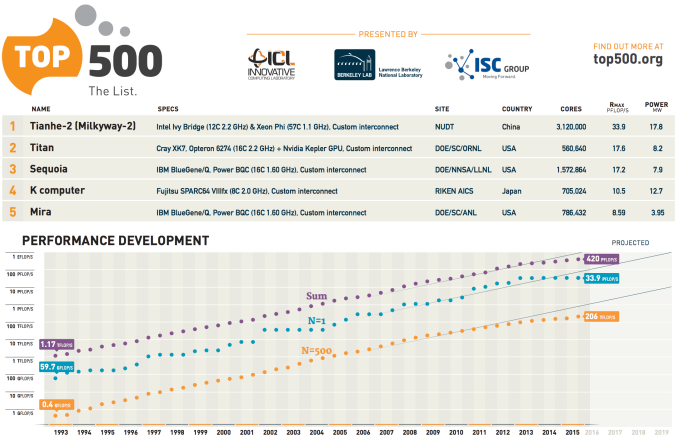


Figure 2: TOP500-poster1

Top500 (2/2)

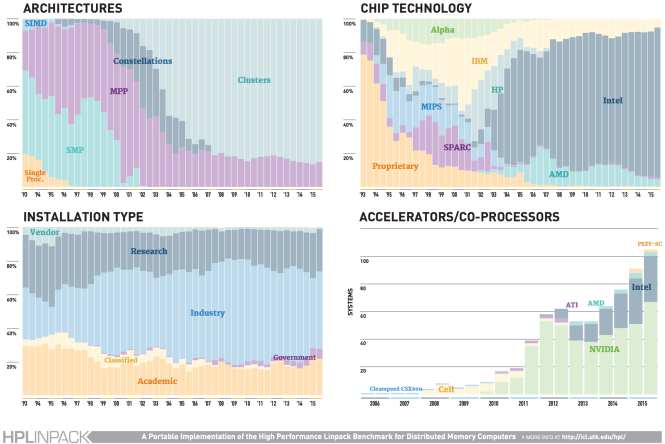


Figure 3: TOP500-poster2

HPC プログラミング

コンピュータの性能評価

FLOPS

- Floating Point Operations Per Second
- 1 秒間に浮動小数点演算 (Floating Point Operations) が何回実行できるか
  - 理論 FLOPS = クロック周波数 x コア数 x クロックあたりの浮動小数点演算数
  - クロック周波数: 1 秒あたりの処理回数
  - 例えば iMac (Intel Core i5 2.8 GHz Quad-core)
    - 2.8 GHz x 4 core x 16 op = 179.2 GFLOPS

- 様々な CPU のクロックあたりの浮動小数点演算数

CPU		備考
Core2 Duo	4 FLOPS/Clock	SSE
Core2 Quad	4 FLOPS/Clock	SSE
Core i7(Nehalem)	4 FLOPS/Clock	SSE
Core i7(SandyBridge)	8 FLOPS/Clock	AVX
Core i7(Haswell ∼)	16 FLOPS/Clock	AVX2
AMD Opteron(Magny-Cours	4 FLOPS/Clock	
AMD FX(Bulldozer)	8 FLOPS/Clock	

- 様々なハードの浮動小数点演算能力

名称			備考
GeForce GTX 480	1401 MHz x 480 core x 2	1.345 TFLOPS	GPU
GeForce GTX TITAN	876 MHz x 2688 core x 2	4.7 TFLOPS	GPU
Cell		218 GFLOPS	PS3(全体:2TFLOPS)
Apple A7	400 MHz x 4 x 64	102.4 GFLOPS	iPhone5s
京		10.51 PFLOPS	
地球シミュレータ		35.86 TFLOPS	
Deep Blue		11.38 GFLOPS	1997

メモリバンド幅

- 単位時間あたりに転送できるデータ量
  - 理論バンド幅 = DRAM クロック周波数 x 1 クロックあたりのデータ転送回数 x メモリバンド幅 (8 byte) x CPU メモリチャンネル数
    - DDR3-1600 なら
      (DRAM クロック周波数 x 1 クロックあたりのデータ転送回数) = 1600
  - 例えば iMac (Intel Core i5-5575R, DDR3)
    - 1867 MHz x 8 x 2 = 29872 MB/s = 29.9 GB/s
  - 計算ノード間 (FX10; 単方向): 20 GB/s
- 単純な計算を大量に行う場合は、メモリバンド幅が性能を決める

Byte per FLOPS

- 通称 B/F 値
- 1 回の浮動小数点演算の間にアクセスできるデータ量
  - FX10: 85 GB/s / 236.5 GFLOPS = 0.36
  - SR16000: 512 GB/s / 980.48 GFLOPS = 0.52
- 参考
  - 倍精度実数 (double) は 8 byte:
    3 度の読み書き (e.g. c=a*b) で 8 x 3 = 24 byte
    - B/F 値 24 以上必要
    - 0.36 / 24 = 0.015 = 1.5% (つまり 98.5% CPU は遊んでる)
- 高速化のためには、如何に CPU を有効活用するかがポイント

階層メモリ構造

名称	記憶容量	アクセス速度 (遅延)	転送速度 (帯域)
レジスタ (on CPU)	byte	ns	GB/s
キャッシュ (on CPU)	kB ~ MB	10 ns	GB/s
(メイン) メモリ	MG ~ GB	100 ns	100 MB/s
ハードディスク	GB ~ TB	10 ms	100 MB/s

- キャッシュを効率的に使わないと遅い

データ格納構造

- データはまとまって取り扱われる (=キャッシュライン)
  - 連続したデータは近く (キャッシュ内) に存在する確率が高い
    - キャッシュヒット
  - 不連続データアクセスはキャッシュミスを引き起こしやすい
- (C/C++言語の)1 次元配列は連続データ
  - うまく活用することで高速化が期待できる

行列積でのメモリアクセス

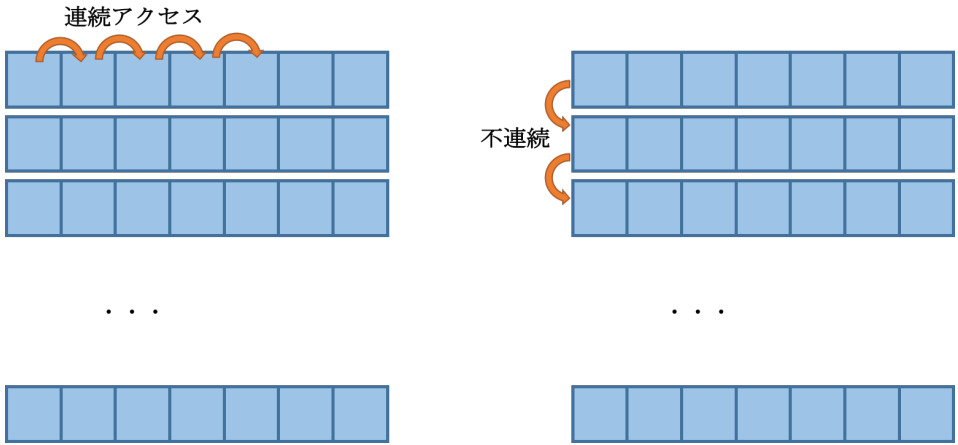


Figure 4: 行列積でのメモリアクセス

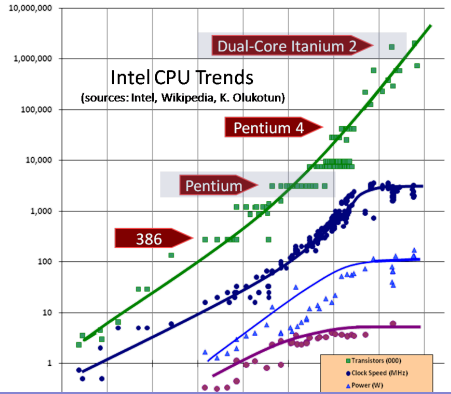
単体チューニング

- CPU へ如何にうまくデータを送り込ませるかがポイント
- 転送量 < 演算量 の場合
  - データを使いまわして高速化 → ブロック化
  - 例： 行列積
    - データ量 N^2
    - 演算量 N^3
- 転送量 > 演算量 の場合
  - 高速化は難しい
    - 余計に計算する (メモリ転送量を減らす) ことも一考
  - 例： 行列とベクトルの積
  - 例： ハウスホルダー三重対角化
    - 行列-ベクトル積が必要 → 帯行列にする

並列化プログラミング

なぜ並列化が必要なのか

- “The Free Lunch Is Over (フリーランチは終わった)”
  - http://www.gotw.ca/publications/concurrency-ddj.htm



フリーランチは終わった

- クロックが上がるとソフトウェアのパフォーマンスも勝手に向上
- クロック上昇の限界
- CPU を複数使用するしかない
- 並列処理のプログラムを書かねばパフォーマンスが上がらず

並列化プログラミングの心構え

- 本当に並列化が必要か
  - まずは単体動作でのチューニングをすべき
  - そもそも単体動作で正しく動くことを確認すること
- どこを並列化すべきか
  - パレートの法則 (80:20 の法則)
  - プロファイラ等を使い、どの関数・ループが処理に時間がかかるかを見つける
  - 思い込みは禁物
- 並列化したらなんでも速くなると思ったら大間違い

(並列) 性能評価指標

台数効果 (高速化率)

$$S_P = \frac{T_S}{T_P}$$

- T_S：1 台 (serial) での実行時間
- T_P：複数台 (P 台; parallel) での実行時間
- どれだけ早く計算できるようになったかを示す指標
  - $S_P = P$ が理想的 (多くは $S_P < P$)
  - $S_P > P$ は super linear speedup とよばれる
  - キャッシュヒットなどによって高速化されたケースなど

並列化効率

$$E_P = \frac{S_P}{P} x100$$

- 並列化がどれだけ上手に行われているかを示す指標

アムダールの法則 Amdahl's law

- 1 台での実行時間 T_S のうち、並列化ができる割合 (並列化率) を a とすると、P 台での並列実行時間 T_P は

$$T_P = \frac{T_S}{P} \cdot a + T_S(1-a)$$

従って台数効果は

$$S_P = \frac{T_S}{T_P} = \frac{1}{(a/P + (1-a))}$$

- 無限台使っても (P →∞), 台数効果は $1/(1-a)$ しか出ない
  - ←アムダールの法則
- 全体の 90%を並列化しても、1/(1-0.9)=10 倍で飽和する

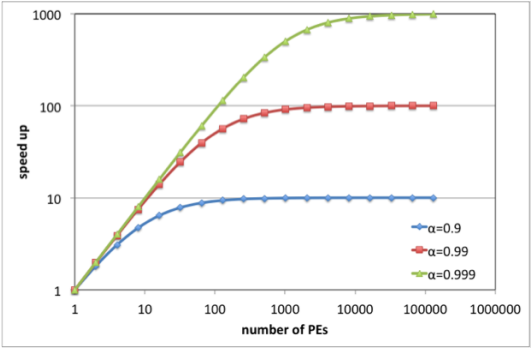


Figure 6: Amdahl's law

アムダールの法則のポイント

- “並列化出来る処理” と “頑張っても並列化できない処理” とがある

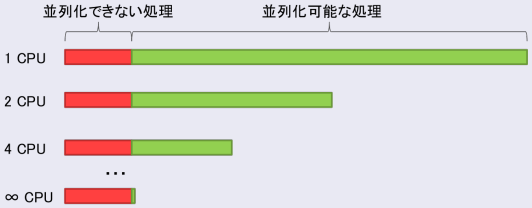


Figure 7: Amdahl_point

スケーラビリティ (並列性能向上) の評価

Excel シートの使い方

プロセスとスレッド

- プロセス
  - OS から独立したリソースを割り当てられる
    - CPU
    - メモリ空間
  - 1 つ以上のスレッドを持つ
  - 親 (プロセス) - 子 (スレッド)
- スレッド
  - 実行単位
  - 各スレッドはプロセス内メモリを共有する

並列プログラミングの仕組みと方法

- マルチプロセス
  - プロセス間でデータのやりとりをする仕組み
  - プロセス間でメモリ空間は (基本的には) 共有できない
  - 別の計算機上にあるプロセスとも通信できる
  - MPI(Message Passing Interface)
- マルチスレッド
  - プロセス内部で複数スレッドが並列動作
  - プロセスのメモリ空間を複数スレッドで共有できる
  - 排他処理が必要
  - 同一システム上でしか動作しない
  - pthread(POSIX thread), OpenMP

MPI の特徴

- ライブラリ規格の一つ
  - プログラミング言語、コンパイラに依存しない
  - API(Application Programing Interface) を標準化
  - 実装がまちまち
- 大規模計算が可能
  - ネットワークを介したプロセス間通信が可能
- プログラミングの自由度が高い
  - 通信処理をプログラミングすることで最適化が可能
  - 裏を返せばプログラミングが大変

MPI の実装

- MPICH
  - Argonne National Laboratory で開発
  - MPICH1, MPICH2 など
- OpenMPI
  - オープンソース
  - 最近の Linux ディストリビューションで採用されつつある
- ベンダー製 MPI
  - 計算機用に最適化された MPI
  - MPICH2 がベースが多い

MPI プログラミングの作法

- 初期化
  - 使う資源 (リソース) を確保・準備する
  - すべてのプロセスが呼び出す必要がある
  - MPI_Init() 関数
- 後始末
  - 使った資源 (リソース) を返す
  - 返さないとゾンビ (ずっと居残るプロセス) になる場合も
  - すべてのプロセスが呼び出す必要がある
  - MPI_Finalize() 関数

MPI 関数の性質

- 通信
  - 集団通信
    - 全プロセスが通信に参加
    - 全プロセスが呼ばなければ止まる
  - 1 対 1 通信
    - 通信に関与するプロセスのみが関数を呼ぶ
- ブロッキング
  - ブロッキング通信
    - 通信が完了するまで次の処理を待つ
  - ノンブロッキング通信
    - 通信しながら別の処理が可能

主な MPI 関数

MPI_Init

```
#include <mpi.h>
int MPI_Init(int *argc, char ***argv);
```

- MPI 環境を起動・初期化する
- パラメータ
  - argc: コマンドライン引数の総数
  - argv: 引数の文字列を指すポインタ配列
- 戻り値: MPI_Success(正常)

MPI_Finalize

```
#include <mpi.h>
int MPI_Finalize();
```

- MPI 環境の終了処理を行う

MPI_Comm_size

```
#include <mpi.h>
int MPI_Comm_size(MPI_Comm comm, int *size);
```

- コミュニケータに含まれる全プロセスの数を返す
- コミュニケータには全 MPI プロセスを表す定義済みコミュニケータ MPI_COMM_WORLDが使用できる
- パラメータ
  - comm: (in) コミュニケータ
  - size: (out) プロセスの総数
- 戻り値: MPI_Success(正常)

MPI_Comm_rank

```
#include <mpi.h>
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

- コミュニケータ内の自身のプロセスランクを返す
  - ランクは 0 から始まる
- パラメータ
  - comm: (in) コミュニケータ
  - rank: (out) ランク
- 戻り値: MPI_Success(正常)

MPI_Bcast

```
#include <mpi.h>
int MPI_Bcast(void* buf, int count, MPI_Datatype datatype,
int root, MPI_Comm comm);
```

- root から comm の全プロセスに対して broadcast する
- パラメータ
  - buf: (in) 送信バッファのアドレス
  - count: (in) 送信する数
  - datatype: (in) データ型
  - root: (in) 送信元ランク
  - comm: (in) コミュニケータ
- 戻り値: MPI_Success(正常)

MPI_Allreduce

```
#include <mpi.h>

int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

- 集計した後、結果を全プロセスへ送信する
- パラメータ
  - sendbuf: (in) 送信バッファのアドレス
  - recvbuf: (in) 受信バッファのアドレス
  - count: (in) 送信する数
  - datatype: (in) データ型
  - MPI_Op: (in) 演算オペレータ
  - comm: (in) コミュニケータ
- 戻り値: MPI_Success(正常)

MPI_Send

```
#include <mpi.h>

int MPI_Send(void* buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm);
```

- dest プロセスへデータを送る
- パラメータ
  - buf: (in) 送信バッファのアドレス
  - count: (in) 送信する数
  - datatype: (in) データ型
  - dest: (in) 送信先ランク
  - tag: (in) タグ
  - comm: (in) コミュニケータ
- 戻り値: MPI_Success(正常)

MPI_Recv

```
#include <mpi.h>

int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status* status);
```

- source プロセスからのデータを受け取る
- パラメータ
  - buf: (in) 送信バッファのアドレス
  - count: (in) 送信する数
  - datatype: (in) データ型
  - source: (in) 送信元ランク
  - tag: (in) タグ
  - comm: (in) コミュニケータ
  - status: (out) ステータス情報

MPI_Isend

```
#include <mpi.h>

int MPI_Isend(void* buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request* request);
```

- dest プロセスへデータを送る
- パラメータ
  - buf: (in) 送信バッファのアドレス
  - count: (in) 送信する数
  - datatype: (in) データ型
  - dest: (in) 送信先ランク
  - tag: (in) タグ
  - comm: (in) コミュニケータ
  - request: (out) リクエストハンドル

MPI_Irecv

```
#include <mpi.h>

int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Request* request);
```

- source プロセスからのデータを受け取る
- パラメータ
  - buf: (in) 送信バッファのアドレス
  - count: (in) 送信する数
  - datatype: (in) データ型
  - source: (in) 送信元ランク
  - tag: (in) タグ
  - comm: (in) コミュニケータ
  - request: (out) リクエストハンドル

MPI_Wait

```
#include <mpi.h>

int MPI_Wait(MPI_Request* request, MPI_Status* status);
```

- 同期待ち処理を行う
- パラメータ
  - request: (in) リクエストハンドル
  - status: (out) 受信状態

MPI データ型

C/C++ data type	MPI data type
char	MPI_CHAR
int	MPI_INT
long	MPI_LONG
float	MPI_FLOAT
double	MPI_DOUBLE
unsigned char	MPI_UNSIGNED_CHAR
unsigned int	MPI_UNSIGNED_INT
unsigned long	MPI_UNSIGNED_LONG

MPI サンプルコード (1/2)

```
#include <iostream>
#include <unistd.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    int rank = 0;
    int size = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

MPI サンプルコード (2/2)

```
    char hostname[256];
    for (int i = 0; i < size; ++i) {
        if (i == rank) {
            gethostname(hostname, sizeof(hostname));
            std::cout << "rank=" << i << ", hostname=" << hostname << std
                ::endl;
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

MPI プログラミングのコツ

- コンパイラは専用のもの (mpicxx, mpifort など) を使う
  - コンパイル・ビルドに必要なライブラリやインクルードパスを自動的に設定してくれる
- 実行は実装によって異なる
  - mpirun? mpiexec?
  - 実装毎に環境変数も変わる
- 基本的にデバッグは難しい
  - 逐次 (シリアル) 版でバグは潰しておく
  - デバッガに頼らず、何かに出力するようにした方が無難
  - gdb オプションも時には使える
- プロファイル
  - gprof なら GMON_OUT_PREFIX 環境変数を使うと良い

MPI 補足

- MPI もソフトウェア
  - バグは少なからずある
  - なるべく実績のある (よく使われる)API を使う
- MPI-1 を使った方が良い (場合がある)
  - MPI-2 以上は多機能な反面、システムによって挙動が異なる場合がある
  - MPI-1 で (やりたいことは) 基本的に実現可能
    - 可変長配列を送るときは、はじめに配列数を転送するなど工夫する。
- 非同期通信が必ずしも良いとは限らない
  - デバッグ作業は格段に難しくなる
  - MPI_Test(), MPI_Wait() が呼ばれて初めて通信を開始する実装がある

OpenMP

OpenMP の特徴

- C/C++および Fortran プログラミング言語をサポートする API(Application Program Interface)
  - 指示文 (pragma なので非対応コンパイラでも問題なし)
  - 専用のライブラリをリンク
  - 環境変数で動作を制御
- 共有メモリ型並列計算機上で動作する
- 並列処理する箇所を明示する必要がある
  - 自動並列化ではない
- データ分割を指示しなくても良い
  - プログラミングが楽
  - 裏を返せば、処理がブラックボックス化
- 最近は GPU コードも吐けるように

OpenMP の書き方 (C/C++)

- 並列実行

```
#pragma omp parallel
{
…
}
```

- 並列実行 (for ループ)

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
…
#pragma omp critical (name) ← クリティカルリージョン
{
  …
}
```

OpenMP サンプル (1/2)

```
#include <iostream>
#include <omp.h>

int main()
{
    std::cout << "# of procs: " << omp_get_num_procs() << std::endl;
    std::cout << "max threads: " << omp_get_max_threads() << std::endl;

#pragma omp parallel
    {
        int id = omp_get_thread_num();
        std::cout << "thread #: " << id << std::endl;
    }
```

OpenMP サンプル (2/2)

```
    int sum = 0;
#pragma omp parallel for
    for (int i = 0; i < 10000; ++i) {

#pragma omp atomic
        sum += i;
    }

    std::cout << "sum=" << sum << std::endl;

    return 0;
}
```

代表的な OpenMP pragma

- ブロックを並列化

```
#pragma omp parallel
{
...
}
```

- for ループを並列化 (1; for を分割処理)

```
#pragma omp parallel
{
#pragma omp for
for (int i = 0; i < 100; ++i) {
...
}
}
```

- for ループを並列化 (2; parallel と一緒に指定)

```
#pragma omp parallel for
for (int i = 0; i < 100; ++i) {
...
}
```

- section を並行に実行

```
#pragma omp parallel sections
{
#pragma omp section
    {
    ...
    }

#pragma omp section
    {
    ...
    }
}
```

- 1 つのスレッドだけが実行

```
#pragma omp parallel
{
#pragma omp single
    {
    ...
    }
}
```

- 直後のブロックを排他的に処理

```
#pragma omp parallel
{
#pragma omp critical
    {
    ...
    }
}
```

- スレッドの同期を取る

```
#pragma omp parallel
{
#pragma omp barrier

}
```

- 共有変数のメモリの一貫性を保つ

```
#pragma omp parallel
{
#pragma omp flush
}
```

- ライブラリ関数
  - omp.h をインクルードすること

```
#include <omp.h>
```

関数名	内容
omp_get_num_procs()	プロセッサの数を返す
omp_get_max_threads()	実行可能なスレッドの最大数を取得
omp_get_num_threads()	実行しているスレッド数を取得
omp_get_thread_num()	実行しているスレッド番号を取得

OpenMP の注意点

- ビルド時は多くの場合コンパイルオプションが必要
  - gnu compiler

```
$ gcc -fopenmp
```

- 共有変数か private 変数かを意識すること
  - #omp parallel文の前にある変数は共有変数
- for ループカウンタは符号付き整数
  - OpenMP 3.0 から符号無しも OK
- 環境変数に注意
  - OMP_NUM_THREADS 並列スレッド数を設定する
  - OMP_SCHEDULE 並列動作を指定

ハイブリッド並列

Flat MPI

- ノード間は MPI ノード内も MPI
- ノード内のメモリが共有できない (プロセスあたりのメモリ量が少ない)
- MPI のコードだけを書けばよい

ハイブリッド並列

- ノード間は MPI ノード内は OpenMP
- ノード内メモリをプロセスが占有できる
- 2 種類の並列コードを書かないといけない

FX10 でのハイブリッド並列実行方法

- ノード間並列は MPI で
- ノード内並列は OpenMP で
- FX10 利用の手引
  - http://www.cc.u-tokyo.ac.jp/system/fx10/fx10-tebiki/
  - 8.4.2　バッチジョブ実行例 (5) を参考にすること

参考文献

MPI

- RIST 青山幸也 著 https://www.hpci-office.jp/pages/seminar_text
- P. パチェコ 著, MPI 並列プログラミング ISBN-13: 978-4563015442
- 片桐孝洋 著, スパコンプログラミング入門: 並列処理と MPI の学習 ISBN-13: 978-4130624534

OpenMP

- OpenMP 入門 http://www.isus.jp/article/openmp-special/getting-started-with-openmp/
- 北山 洋幸 著, OpenMP 入門—マルチコア CPU 時代の並列プログラミング ISBN-13: 978-4798023434

FX10 演習環境の構築

概要

- ECCS のマシン (iMac) にログインする
- ターミナルを起動する
  - コンソール画面が表示される
- ssh で FX10 システム (Oakleaf) にログインする

ssh 接続の仕組み

- 暗号化の必要性
  - インターネットにおけるデータの盗聴・なりすましの危険
- 公開鍵方式
  - 秘密鍵で暗号化したデータ → 公開鍵でしか復号できない
  - 公開鍵で暗号化したデータ → 秘密鍵でしか復号できない

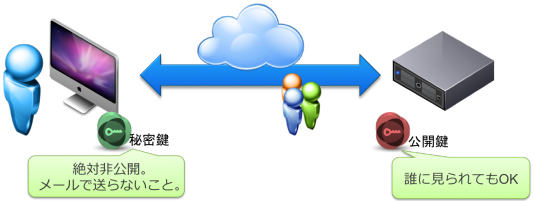


Figure 9: ssh-connection

ssh 鍵の作成

- ターミナルを起動する
- ssh-keygen を実行する

```
$ ssh-keygen -t rsa
```

- 出来るファイル
  - $HOME/.ssh/id_rsa
    - 秘密鍵
    - 誰にも見せないこと
    - メールで送らないこと
  - $HOME/.ssh/id_rsa.pub
    - 公開鍵 (見られても OK)

ssh 公開鍵の登録

- 詳しくは http://www.cc.u-tokyo.ac.jp/system/fx10/fx10-login.html
- 手順
  - web ブラウザ (safari) を立ち上げる
  - 以下の URL を入力する
    - https://oakleaf-www.cc.u-tokyo.ac.jp/cgi-bin/hpcportal/index.cgi
  - アカウントとパスワードを入力する
    - パスワードはそのものではなく、表示されている文字列の奇数番目を繋ぎ合わせたもの
  - 公開鍵を登録する

FX10 へのログイン

- ターミナルから以下を入力

```
$ ssh [FX10のアカウント名]@oakleaf-fx.cc.u-tokyo.ac.jp
```

- パスフレーズが聞かれた場合は、設定したパスフレーズを入れる
- 成功するとログインできる

FX10 とのファイル転送

- scp を使う

```
$ scp [転送元] [転送先]
```

- cp コマンドと同様の使い方 (第 4 文型: SVOO)
  - -r オプションで (サブ) ディレクトリも一緒に
- Mac から FX10 へ @Mac

```
$ scp ./sample.c oakleaf-fx.cc.u-tokyo.ac.jp:somewhere
```

- FX10 から Mac へ @Mac

```
$ scp oakleaf-fx.cc.u-tokyo.ac.jp:sample.c ./somewhere
```

バッチシステムでの実行方法

- 多くのスパコンではインタラクティブな実行はせず、バッチ処理を行う
- FX10 システムでは PJM と呼ばれるバッチシステムを利用
- 使い方

内容	コマンド
ジョブの投入	pjsub “スクリプト”
状況確認	pjstat
混雑度を見る	pjstat -b
ジョブの削除	pjdel “ジョブ ID”
実行中のジョブの削除	pjdel –k “ジョブ ID”