

2017年度 並列・分散システム 第13回

島根大学大学院総合理工学研究科
情報システム領域
浜口清治

木探索の並列化

- ・巡回セールスマン問題を題材に、木探索のスレッド版プログラムの構成を見る。
- ・静的並列化バージョン
 - /u/stu/lecture/hama/Para-Dist-Exercises/ipp-source/ch6/pth_tsp_stat.c
- ・動的並列化バージョン
 - /u/stu/lecture/hama/Para-Dist-Exercises/ipp-source/ch6/pth_tsp_dyn.c

図版などは次の書籍から引用

Peter S. Pacheco : An Introduction to Parallel Programming, Morgan Kaufmann, 2011年.

巡回セールスマン問題

- 問題
 - 全ての都市(ノード)をそれぞれ1度だけ訪れて出発点の都市に戻ってくる.
 - 2つの都市の間の移動(辺)にはコストが定められている.
 - コストの総和が最小のツアー(経路)をみつけるのが目標.
 - この問題は NP完全問題
- NP完全問題
 - 高速の解法は知られていない.

ツアーコスト

- 例

有向グラフで表現

ノードが各都市、

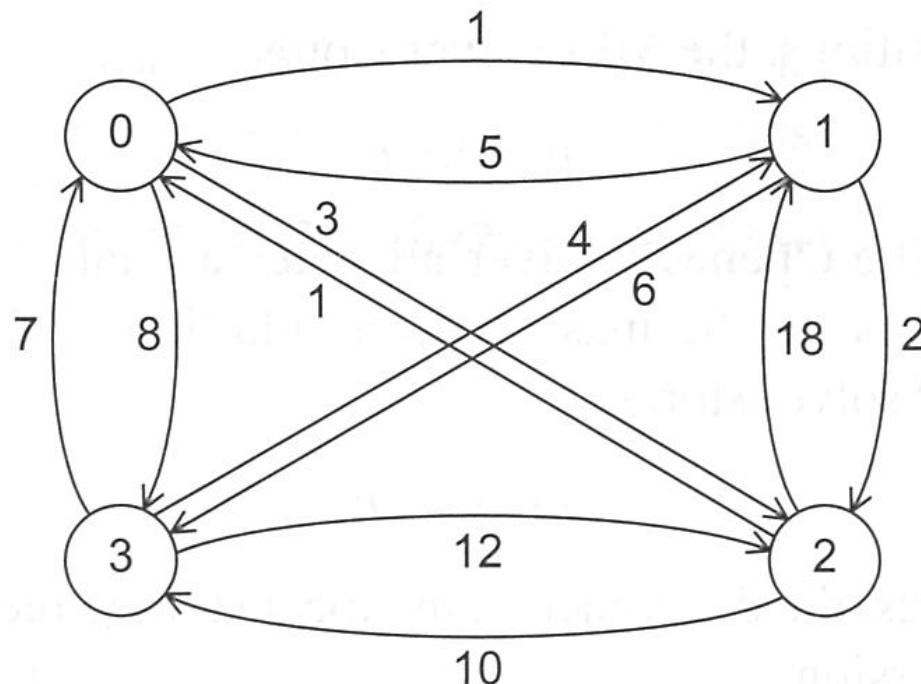
有向辺上のラベルがコスト。

ツアーコストはグラフ上の経路

ツアーコスト $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0$ コスト 20

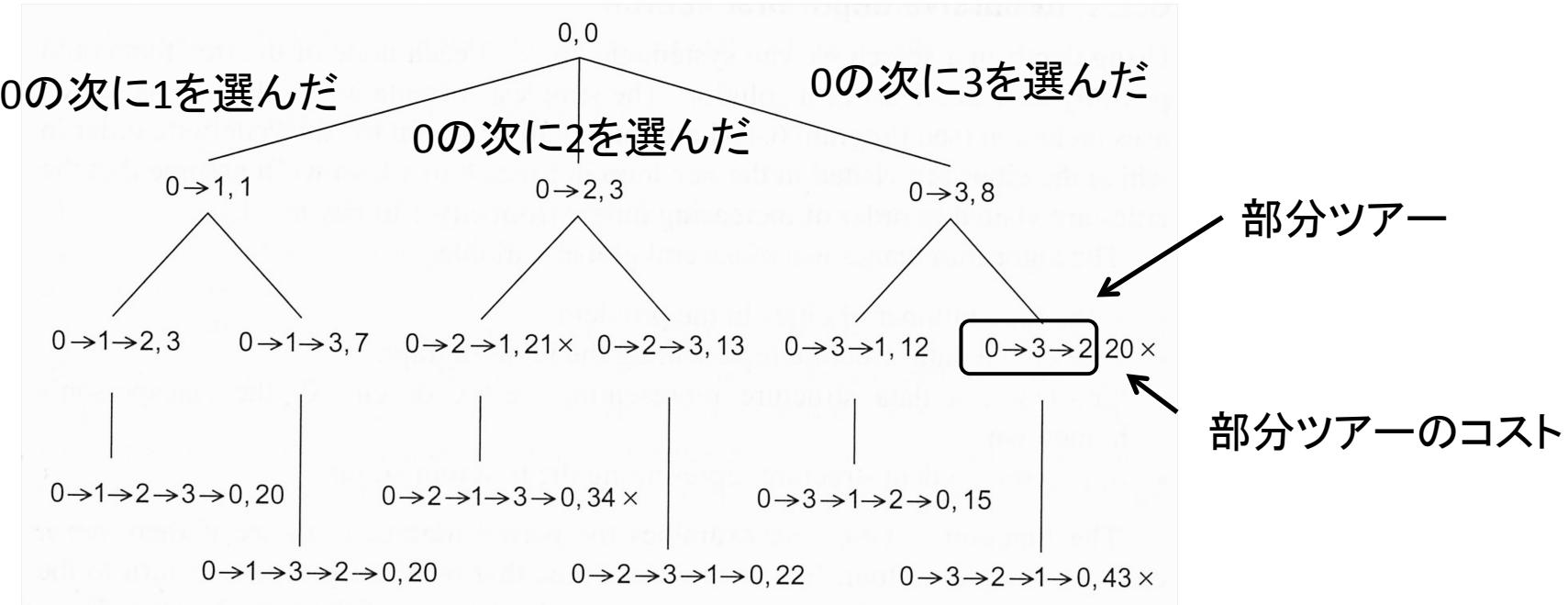
ツアーコスト $0 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 0$ コスト 22

ツアーコスト $0 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 0$ コスト 15



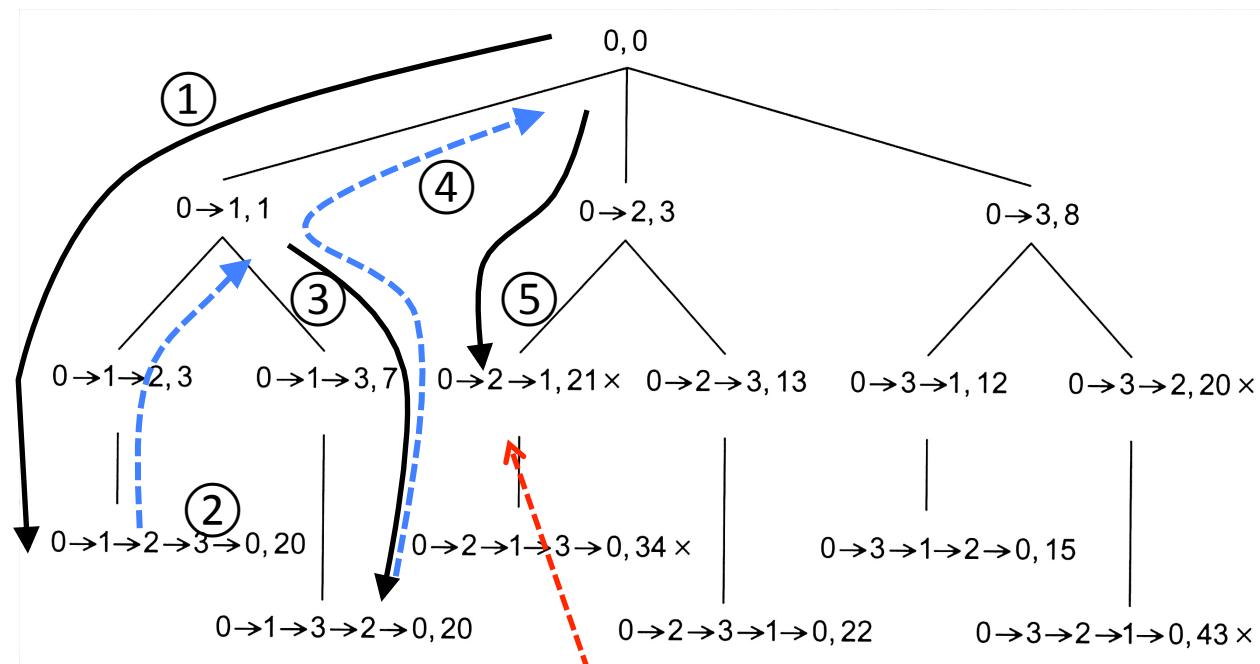
木とツアーノードの対応

- 解を探索する際に木を構成していく



深さ優先探索

- 深さ優先で探索する



実は、この節点からは
コスト20以下のツアーはみつからない
ことがわかるので、ここで止めてよい

深さ優先探索

葉に到達するか、
最小コストのツアーニ
到達することができない
節点に到達



まだ探索していない子を
持つ「先祖」のノードへ
戻って(バックトラックして)
未探索の子を選んで
探索を続ける。

深さ優先探索アルゴリズム

- 再帰呼び出し版

```
1 void Depth_first_search(tour_t tour)
2     city_t city;
3
4     if (City_count(tour) == n) {
5         if (Best_tour(tour))
6             Update_best_tour(tour);
7     } else {
8         for each neighboring city
9             if (Feasible(tour, city)) {
10                 Add_city(tour, city);
11                 Depth_first_search(tour);
12                 Remove_last_city(tour);
13             }
14     }
15 } /* Depth_first_search */
```

tour : ツアーを格納
(例 : $0 \rightarrow 1 \rightarrow 2$)

City_count : ツアー内の都市の数を返す

n : 都市の総数

Best_tour : ここまでで最小コストか?

Update_best_tour : tour を best_tour に設定

tour の最後の都市に隣接する各 city について

Feasible (tour,city) : city が tour で未訪問か,
あるいはそれ以上探索しても最小コストのツ
アーに到達しないか (best_tour と比べる)?

Add_city(tour, city) : tour の最後に
city を付け加える

Depth_first_search : 再帰呼び出し

Remove_last_city: tour から city を
取り除き, tour を縮める.

深さ優先探索アルゴリズム

- 再帰呼び出し
 - 再帰呼び出しは一般に速度が遅い.
 - 1つの時点では、木の1つの節点にのみにしかアクセスできないので、並列化が難しい.
- 再帰呼び出しの除去方法
 - 再帰呼び出しにおける現在の状態(都市)を、スタックにpush
 - 先祖の方向にバックトラックするときには、スタックからpop

非再帰版の深さ優先探索(1)

- スタックに置く情報: 次の2種類
 - city : tour の一番最後に加えられた都市
 - NO_CITY : 定数値
 - 現在の木の節点からたどることができる子の節点については、すでに探索が終わっていることを示すためのマーカーの役割をはたす

非再帰版の深さ優先探索(1)

```
1 for (city = n-1; city >= 1; city—)
2     Push(stack, city);
3 while (!Empty(stack)) {
4     city = Pop(stack);
5     if (city == NO_CITY) // End of child
6         Remove_last_city(curr_tour);
7     else {
8         Add_city(curr_tour, city);
9         if (City_count(curr_tour) == n) {
10             if (Best_tour(curr_tour))
11                 Update_best_tour(curr_tour);
12             Remove_last_city(curr_tour);
13         } else {
14             Push(stack, NO_CITY);
15             for (nbr = n-1; nbr >= 1; nbr—)
16                 if (Feasible(curr_tour, nbr))
17                     Push(stack, nbr);
18         }
19     } /* if Feasible */
20 } /* while !Empty */
```

city を n-1 から 1まで stack に push
stack が空になるまで、次を繰り返す
stack から pop した値が NO_CITY のときは、現在の木の節点の子はすべて探索済みなので、ひとつ上の節点へ戻る(=curr_tour を縮める)
stack から pop した値が city なら、Add_city によって curr_tour に city を付け加える。
すべての都市を訪問済みなら、curr_tour が最小コストかどうか調べて、そうなら、best_tour に登録。curr_tour から最後の city を削除
未訪問の都市があれば、NO_CITY をスタックにpush。その後、都市 (nbr) を stack にpushする

非再帰版の深さ優先探索(1)

```
curr_tour = 0
1 for (city = n-1; city >= 1; city--)
2     Push(stack, city);
3 while (!Empty(stack)) {
4     city = Pop(stack);
5     if (city == NO_CITY) // End of child list, back up
6         Remove_last_city(curr_tour);
7     else {
8         Add_city(curr_tour, city);
9         if (City_count(curr_tour) == n) {
10            if (Best_tour(curr_tour))
11                Update_best_tour(curr_tour);
12            Remove_last_city(curr_tour);
13        } else {
14            Push(stack, NO_CITY);
15            for (nbr = n-1; nbr >= 1; nbr--)
16                if (Feasible(curr_tour, nbr))
17                    Push(stack, nbr);
18        }
19    } /* if Feasible */
20} /* while !Empty */
```

The diagram illustrates the state of variables across 20 iterations of the algorithm. It uses arrows to point from specific lines of code to their corresponding variable values at that iteration.

- Iteration 1:** curr_tour = 0
- Iteration 2:** stack = 1, 2, 3
- Iteration 4:** city = 1, stack = 2, 3
- Iteration 8:** curr_tour = 0 → 1
- Iteration 15:** stack = NO_CITY, 2, 3
- Iteration 18:** stack = 2, 3, NO_CITY, 2, 3

非再帰版の深さ優先探索(2)

- 並列化のために別のバージョンを考える
- スタックの内容が都市(ノード)1つ1つだと、取りだしたとき、木のどの部分を探索しているかわからない。
- スタックに、都市の代わりに(部分)ツリーをいれておくと、木のどの部分を処理しているかわかる。
- その後の処理を独立して実行できるので、並列化する際に有利。
 - ただし、このバージョンは、部分ツリーをスタックへプッシュする時にコピーを作るので、並列化しない場合はオーバヘッドが大きいと予想できる。

非再帰版の深さ優先探索(2)

```
1 Push_copy(stack, tour); // Tour that visits only the hometown
2 while (!Empty(stack)) {
3     curr_tour = Pop(stack);
4     if (City_count(curr_tour) == n) {
5         if (Best_tour(curr_tour))
6             Update_best_tour(curr_tour);
7     } else {
8         for (nbr = n-1; nbr >= 1; nbr--)
9             if (Feasible(curr_tour, nbr)) {
10                 Add_city(curr_tour, nbr);
11                 Push_copy(stack, curr_tour);
12                 Remove_last_city(curr_tour);
13             }
14     }
15     Free_tour(curr_tour);
16 }
```

stack = 0

curr_tour = 0, stack = \emptyset

Push_copy (stack, curr_tour) :
curr_tour をコピーして stack に
push する

curr_tour = 0, stack = 0->1,0->2,0->3

データ構造

- スタック：配列で実現する
 - 実はスタック上の要素の個数は $n^2/2$ を越えない。
- Push と Push_copy の疑似コード

```
void Push(my_stack_t stack, int city) {  
    int loc = stack->list_sz;  
    stack->list[loc] = city;  
    stack->list_sz++;  
} /* Push */
```

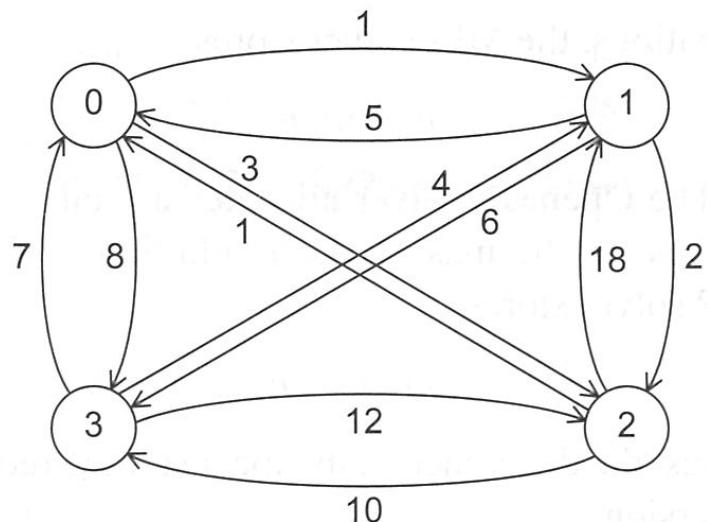
スタックには city を格納

```
void Push_copy(my_stack_t stack, tour_t tour) {  
    int loc = stack->list_sz;  
    tour_t tmp = Alloc_tour();  
    Copy_tour(tour, tmp);  
    stack->list[loc] = tmp;  
    stack->list_sz++;  
} /* Push */
```

スタックには tour のコピーである tmp を格納

データ構造

- 有向グラフ
 - ここでは、隣接行列(adjacency matrix)を使う

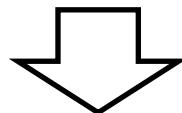


	列	0	1	2	3
行	0	x	1	3	8
1	5	x	2	6	
2	1	18	x	10	
3	7	4	12	x	

例えば 1行0列 の 5 は辺 1->0のコスト
※ 対角要素は使わない

木探索の並列化(1)

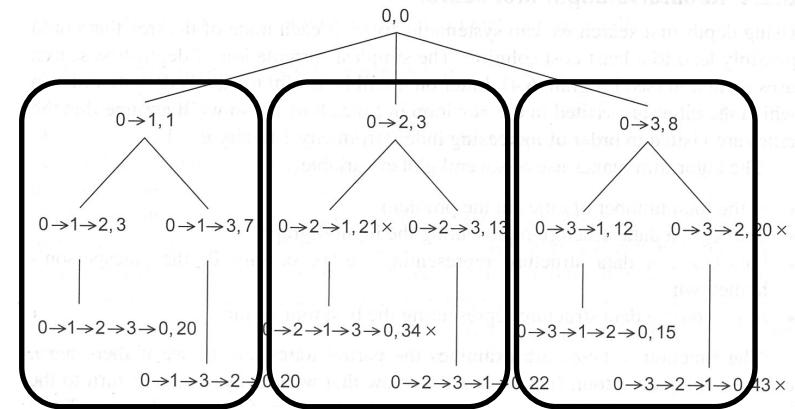
- 基本方針
 - 木のそれぞれのノードにタスクを割り当てる
 - そのノードでの部分ツア－ curr_tour がそこまでのベストの解を越えられる見込みがないなら打ち切り
 - 可能性があれば, city を curr_tour に加えて子のノードを作る



- 異なるノードに対するタスクを並列に実行する

木探索の並列化(2)

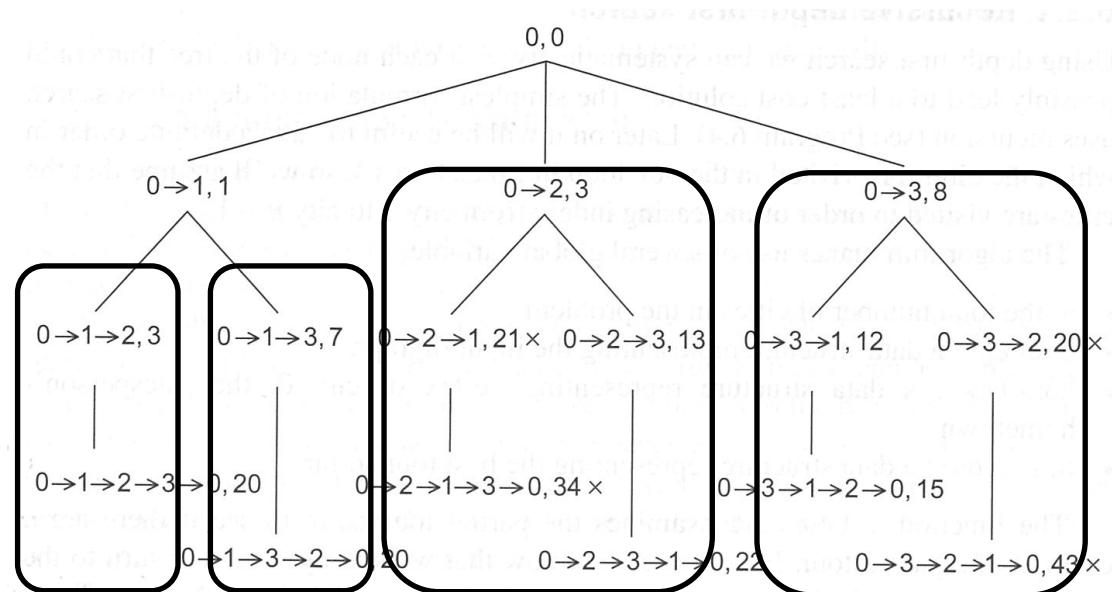
- 注意しなければならない点
 - best_tour の更新
 - 並列に処理しているスレッドに知らせる必要がある.
 - 排他制御も必要
- 自然なタスクの割当て方法
 - 部分木を別のスレッドに割り当てる



別々のスレッドに

木探索の並列化(3)

- 部分木への分割
 - 幅優先探索で節点の展開を行って、スレッド/プロセス数に達したところで、部分ツリーを各スレッド/プロセスへ渡す

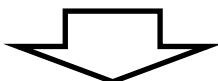


スレッド数が4の場合の分け方の一例

タスクの割当て- 静的/動的

1. 静的なタスクの割当て

- 各スレッドの担当する部分木を最初に固定的に決める
- 部分木ごとに計算処理の量がばらつく可能性がある



2. 動的なタスクの割当て

- あるスレッドのタスクが尽きてしまったら(終わってしまった), 他のスレッドのタスクを分けてもらう.
 - ・具体的にはスタックに格納している内容(部分ツリー)を分割する.

pthread による静的な並列化

1. 幅優先探索で部分ツアーを生成 (この資料では省略)
2. 並列化して深さ優先探索

各スレッドで実行する部分(スレッド生成)の疑似コード

```
Partition_tree(my_rank, my_stack);  
  
while (!Empty(my_stack)) {  
    curr_tour = Pop(my_stack);  
    if (City_count(curr_tour) == n) {  
        if (Best_tour(curr_tour)) Update_best_tour(curr_tour);  
    } else {  
        for (city = n-1; city >= 1; city--)  
            if (Feasible(curr_tour, city)) {  
                Add_city(curr_tour, city);  
                Push_copy(my_stack, curr_tour);  
                Remove_last_city(curr_tour)  
            }  
    }  
    Free_tour(curr_tour);  
}
```

このスレッドが
担当する部分ツアーを
my_stack へコピー

その他の部分の流れは、
非再帰版 (2) と同じ

排他制御

- しかし、いくつの関数では排他制御が必要
- Best_tour(curr_tour)
 - その時点までの最良のコストと curr_tour のコストを比較する
 - コストの読み出しと比較だけを行う
 - 読み出した直後に、他のスレッドが最良のコストをアップデートして、読み出した値が古くなる可能性もある
 - しかし、排他制御を導入すると、計算コストが大きくなるので、古い値をそのまま使うことにする
 - つまり、Best_tour については、排他制御は行わない

排他制御

- `Update_best_tour(curr_tour)`
 - `curr_tour` を `best_tour` として登録する
 - 排他制御を行う必要がある.
 - 排他制御を行っていないところで, `Best_tour(curr_tour)` を呼び出した後, 排他制御区間(クリティカルセクション)に入るため, その間に `best_tour` がアップデートされる可能性がある
 - 排他制御区間でもう一度 `Best_tour(tour)`を呼び出して, 確認する.

関数 `Update_best_tour(curr_tour)`

```
pthread_mutex_lock(best_tour_mutex);
/* We've already checked Best_tour, but we need to check it
   again */
if (Best_tour(tour))
    Replace old best tour with tour;
pthread_mutex_unlock(best_tour_mutex).
```

pthreadによる動的な並列化(1)

- 基本的なアイデア
 - タスクがなくて待っている (=スタックが空の)スレッドがあれば、タスクを持っている (=スタックが空でない)スレッドが自分のスタック内の部分ツリーを分けて渡す
- 詳細
 - タスクがなくて待っているスレッド
 - 変数 `thread_in_cond_wait` (待っているスレッドの数)を1増加
 - `pthread` の関数 `pthread_cond_wait` を呼んで待ち状態に入り、信号が送られてくるのを待つ
 - タスクを持っているスレッド
 - 待っているスレッドがあれば、スタックを分割して、待っているスレッドに `pthread` の関数 `pthread_cond_signal` で信号を送る。

pthread_cond_wait と pthread_cond_signal

```
pthread_mutex_t mutex;  
pthread_cond_t cond_var;
```

変数宣言

```
pthread_mutex_lock(&mutex);  
  
pthread_cond_wait(&cond_var, &mutex);  
  
pthread_mutex_unlock(&mutex);
```

pthread_cond_wait は、相互排除区間（危険領域）の中で使う

pthread_cond_wait は信号が来るまで待つ。来るまでは、mutex で指定された相互排除を一時的に解除して、他のスレッドが、危険領域に入ることを許す。

信号がきたら、相互排除を取り戻して、つぎへ進む。

※ 普通 while の中にいれることが多い

```
pthread_mutex_lock(&mutex);  
  
while(pthread_cond_wait(&cond_var, &mutex) != 0);  
  
pthread_mutex_unlock(&mutex);
```

実質的に、次の動作をするのと同じ

```
pthread_mutex_unlock(&q->mutex);  
wait_on_signal(&q->not_full);  
pthread_mutex_lock(&q->mutex);
```

```
pthread_cond_signal(&cond_var);
```

pthread_cond_signal は信号を送って待機状態のスレッドを解除する。
スレッドは1つだけ選ばれる

pthread による動的な並列化(2)

- 停止判定
 - あるスレッドのスタックが空になったとき,
`thread_in_cond_wait == thread_count -1` となれば、すべてのスレッドが仕事を終えたことになる
 - このとき、関数 `pthread_cond_broadcast` を使って、待ち状態にある全てのスレッドを解除して、終了する。
 - `pthread_cond_signal` と異なり、**全てのスレッドの待ち状態を解除する**

関数 Terminated()

- 静的プログラム中の while (!Empty(stack)) の Empty(stack) の代わりに使う

```
1 if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
2     new_stack == NULL) {
3     lock term_mutex;
4     if (threads_in_cond_wait > 0 && new_stack == NULL) {
5         Split my_stack creating new_stack;
6         pthread_cond_signal(&term_cond_var);
7     }
8     unlock term_mutex;
9     return 0; /* Terminated = false; don't quit */
```

自分のスタックに2つ以上要素があり、かつ、待っているスレッドがあり、かつ new_stack が NULL ならば

排他制御を行って、スタックを分けて new_stack に入れて信号を送る

new_stack は分割した stack の中身を置く場所。全スレッドで共有されている。
new_stack の NULL を2度確認しているのは、排他制御区間にはいる前に他のスレッドが new_stack に書き込みを行っている可能性があるため

関数 Terminated()

```
10 } else if (!Empty(my_stack)) /* Keep working */
11     return 0; /* Terminated = false; don't quit */
12 } else { /* My stack is empty */
13     lock term_mutex;
14     if (threads_in_cond_wait == thread_count-1)
15         /* Last thread running */
16     threads_in_cond_wait++;
17     pthread_cond_broadcast(&term_cond_var);
18     unlock term_mutex;
19     return 1; /* Terminated = true; quit */
```

待っているスレッドがなく、
自分のスタックが空でなければ
タスク処理を続ける

自分のスタックが空で、
待っている他のスレッドも
ない場合は、終了

関数 Terminated()

```
20 } else { /* Other threads still working,
21     threads_in_cond_wait++;
22     while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);
23     /* We've been awakened */
24     if (threads_in_cond_wait < thread_count) { /* We got work */
25         my_stack = new_stack;
26         new_stack = NULL;
27         threads_in_cond_wait--;
28         unlock term_mutex;
29         return 0; /* Terminated = false */
30     } else { /* All threads done */
31         unlock term_mutex;
32         return 1; /* Terminated = true; quit */
33     }
34 } /* else wait for work */
35 } /* else my_stack is empty */
```

自分のスタックが空で、他のスレッド
がまだ処理中のとき
待ち状態にはいって、タスクをわけて
もらえるようになるのを待つ

分けてもらえるスレッドがあれば
new_stack からタスク(部分ツアード)
を取り込んで進める

全て終了

関数 pthread_mutex_trylock

- スタックを分割しようとした時、排他制御区間になかなか入れない状況があり得る。
 - 多くのスレッドが排他制御区間にはいろうとしている場合
- 関数 `pthread_mutex_trylock` は、排他制御を一旦試みて、ダメな場合は待たずに `trylock` で指定された区間を読み飛ばす
- これにより待ち時間を短縮できる

```
1 if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
2     new_stack == NULL) {
3     lock term_mutex;
4     if (threads_in_cond_wait > 0 && new_stack == NULL) {
5         Split my_stack creating new_stack;
6         pthread_cond_signal(&term_cond_var);
7     }
8     unlock term_mutex;
9     return 0; /* Terminated = false; don't quit */
```

この部分

スタックの分割

- スタックの分割方法
 - スタックは下から順に部分ツアーの長さが短くなっている。
 - 1つずつpopで取りだして、交互に2つのスタックにpushすれば、長さの均衡のとれた部分ツアーのスタック2つができる。

実験評価

- 実験結果

Table 6.8 Run-Times of Pthreads Tree-Search Programs
(times in seconds)

Threads	First Problem			Second Problem		
	Serial	Static	Dynamic	Serial	Static	Dynamic
1	32.9	32.7	34.7 (0)	26.0	25.8	27.5 (0)
2		27.9	28.9 (7)		25.8	19.2 (6)
4		25.7	25.9 (47)		25.8	9.3 (49)
8		23.8	22.4 (180)		24.0	5.7 (256)

レポート課題 13

※ 以下, best_tour は, その時点で最良の tour を格納している変数であり, 関数 Best_tour でアップデートされる.

- 13-1
 - p.13-5 と p.13-6 の例について, p.13-11 の非再帰版の深さ優先探索(1)を適用する. p.13-6 の①から⑤までに対応する部分を実行した時, curr_tour, city, stack, best_tour の中身はどのように変化していくかを示せ. (ヒント : p.13-11 を参考にする).
- 13-2
 - 13-1と同じ例で p.13-13 の非再帰版の深さ優先探索(2)を適用した場合どうなるか. curr_tour, stack, best_tour の中身の変化を示せ. (ヒント: p.13-13 を参考にする)

※ 特に書き方は問わない. わかるように書かれていればよい.

レポート課題 13

- 13-3
 - p.13-20 と p. 13-22 をみると, `Update_best_tour` の直前で `Best_tour(curr_tour)` がチェックされて, さらに `Update_best_tour` の内部でも `Best_tour(tour)` がチェックされている (`curr_tour` と `tour` は同じ内容).
 - `Update_best_tour` の内部の `Best_tour(tour)` による条件判定を取り除いて, "Replace best tour with tour" を実行するとする.
 - どのようなまずい状況が発生し得るか, 理由とともに説明せよ.
- 13-4
 - p.13- 26 の2行目と4行目をみると, `new_stack == NULL` が 2度 チェックされている. 後の方の `new_stack == NULL` を条件判定から取り除くとどのようなまずい状況が発生し得るか, 理由とともに説明せよ.

レポート課題 14

- 14-1
 - p.13-2 にあげた2つのプログラムを動作させて、性能を比較した結果を示せ
 - 実行方法については、プログラム冒頭のコメント参照。
 - 入力ファイルとして、グラフ表現をあたえる必要がある。プログラムの冒頭のコメントに Input として説明がある。

4	←	都市の数
0 1 3 8		
5 0 2 6		
1 1 8 0 10		
7 4 12 0		

] ← 隣接行列
要素間は空白で区切る
(対角要素は0)

入力ファイルの例