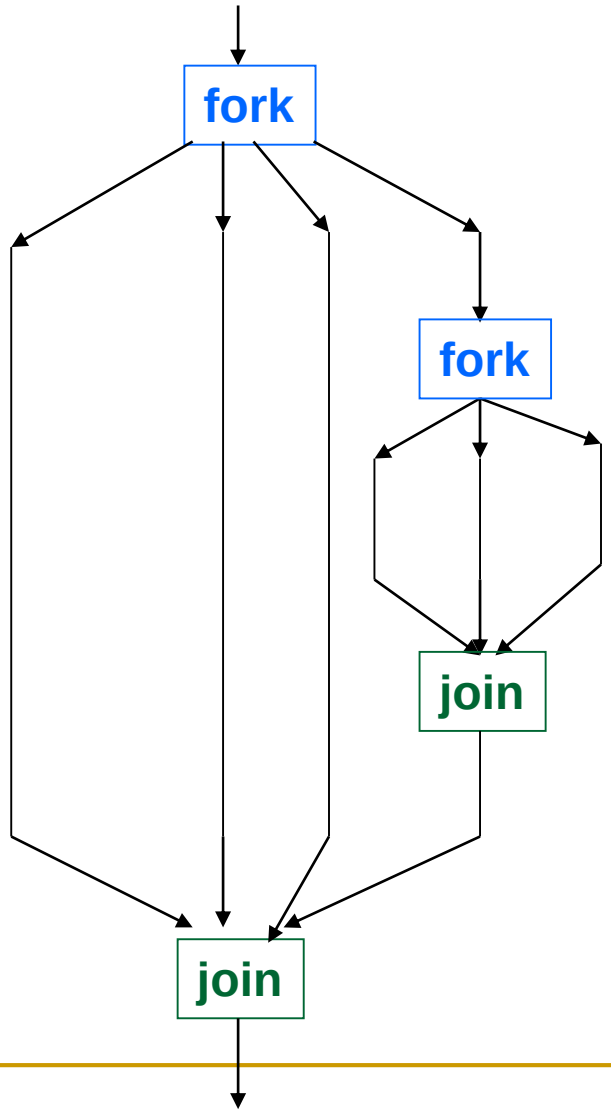
図1 Fork-join: 並列プロセスの開始と終了

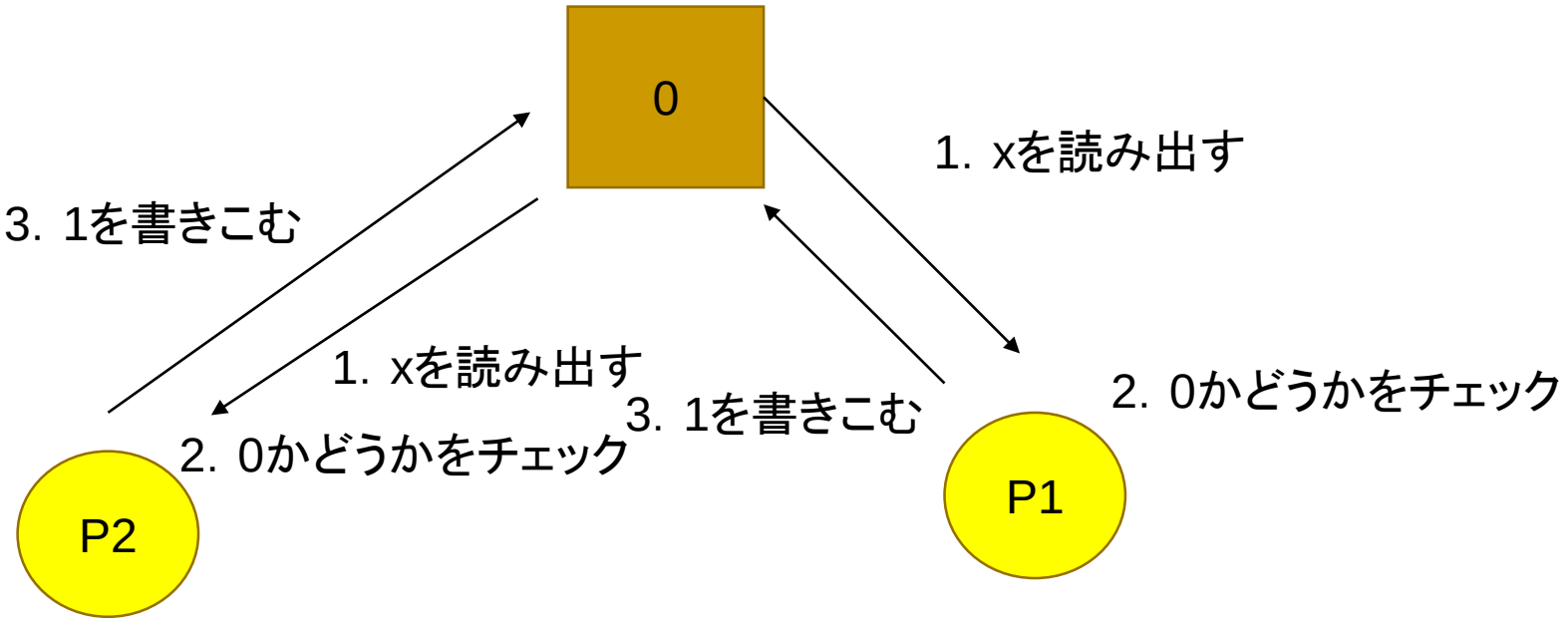


forkで生まれたプロセス（スレッド）間はメモリを共有

joinで全プロセスの待ち合わせをする→同期を取っている

簡単な並列プログラムはfork/joinのみで制御できる→OpenMP

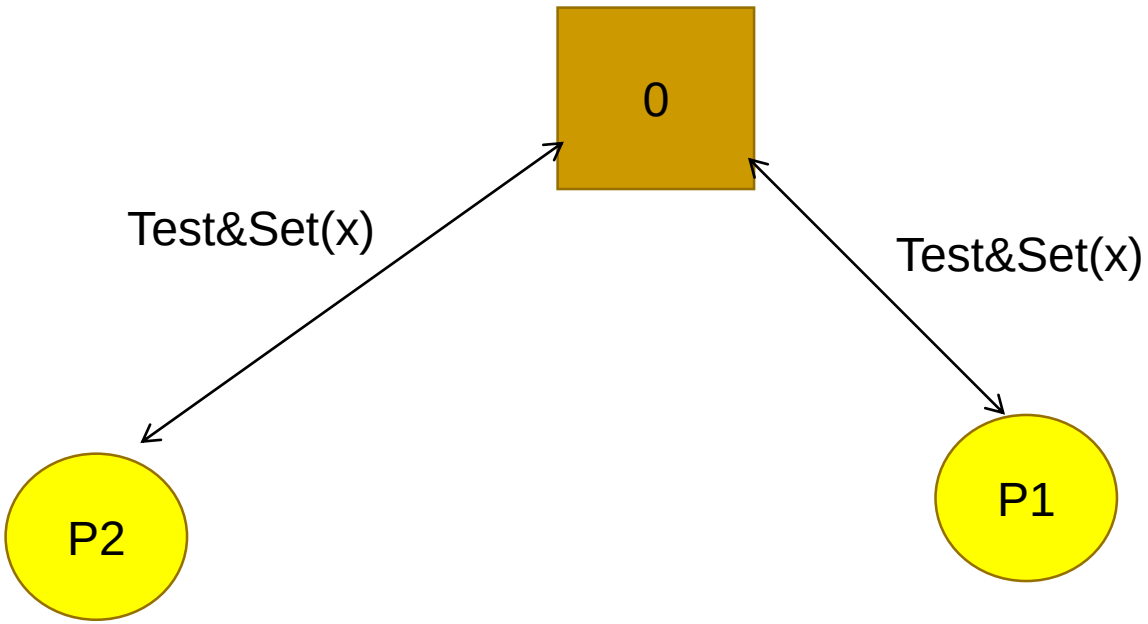
図2a)　P1とP2が同時に変数を読んだら？



P1がxを読んで0かどうかをチェックしている間に、P2がxを読み出すかもしれない→P1,P2共に0を取ることができる。

読む操作と書く操作を不可分（Atomic／Indivisible)に行う命令が必要
→不可分命令

図2b) Test & Set (x)



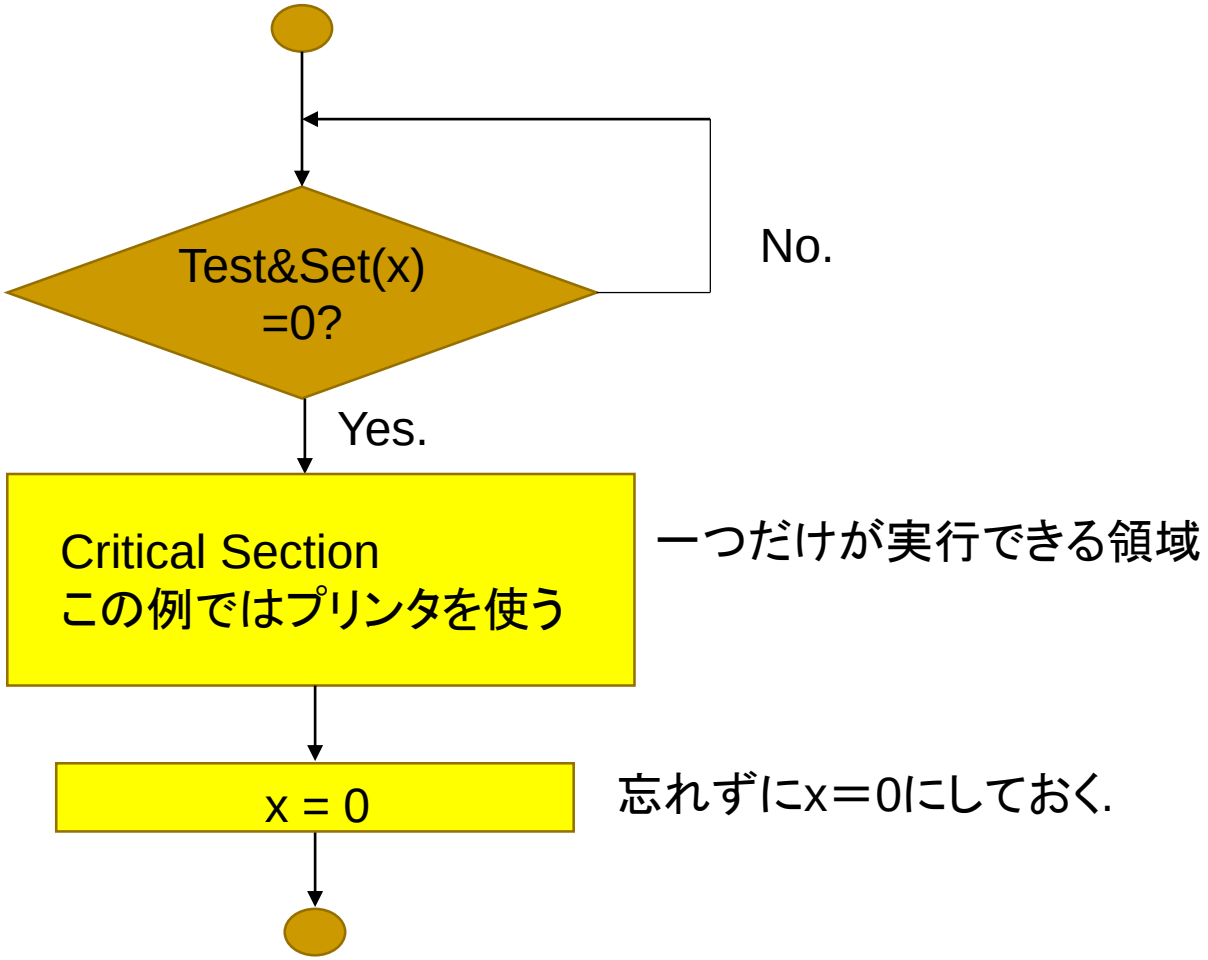
xを読み出す。
1を書きこむ
2つの操作を不可分に行う

この間共有メモリを占有

同時に命令が実行されても、必ず一つだけ0を読み、他は1にする。
→ハードウェアの支援が必要

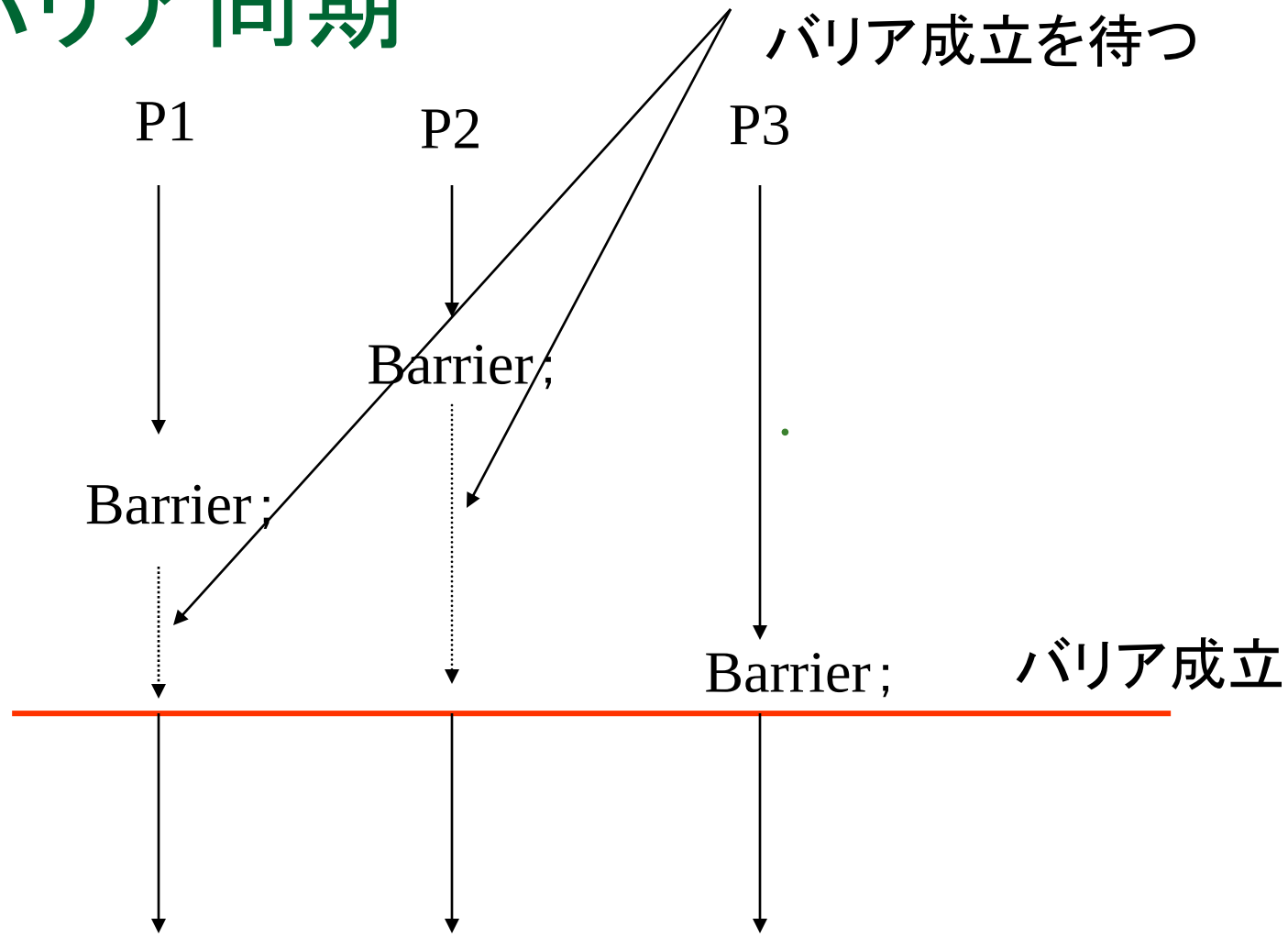
他にもSwap, Compare&Swap, Fetch&Dec, Fetch&Addなど色々あるが、原理は同じ

図3　Critical Sectionの実行



不可分命令があればCritical Sectionが作れる→なんでもできる！
でもちょっと使いにくい

図5バリア同期



バリア同期は不可分命令があれば作れるが、専用のハードウェアを使う場合もある

図6Fuzzy barrier

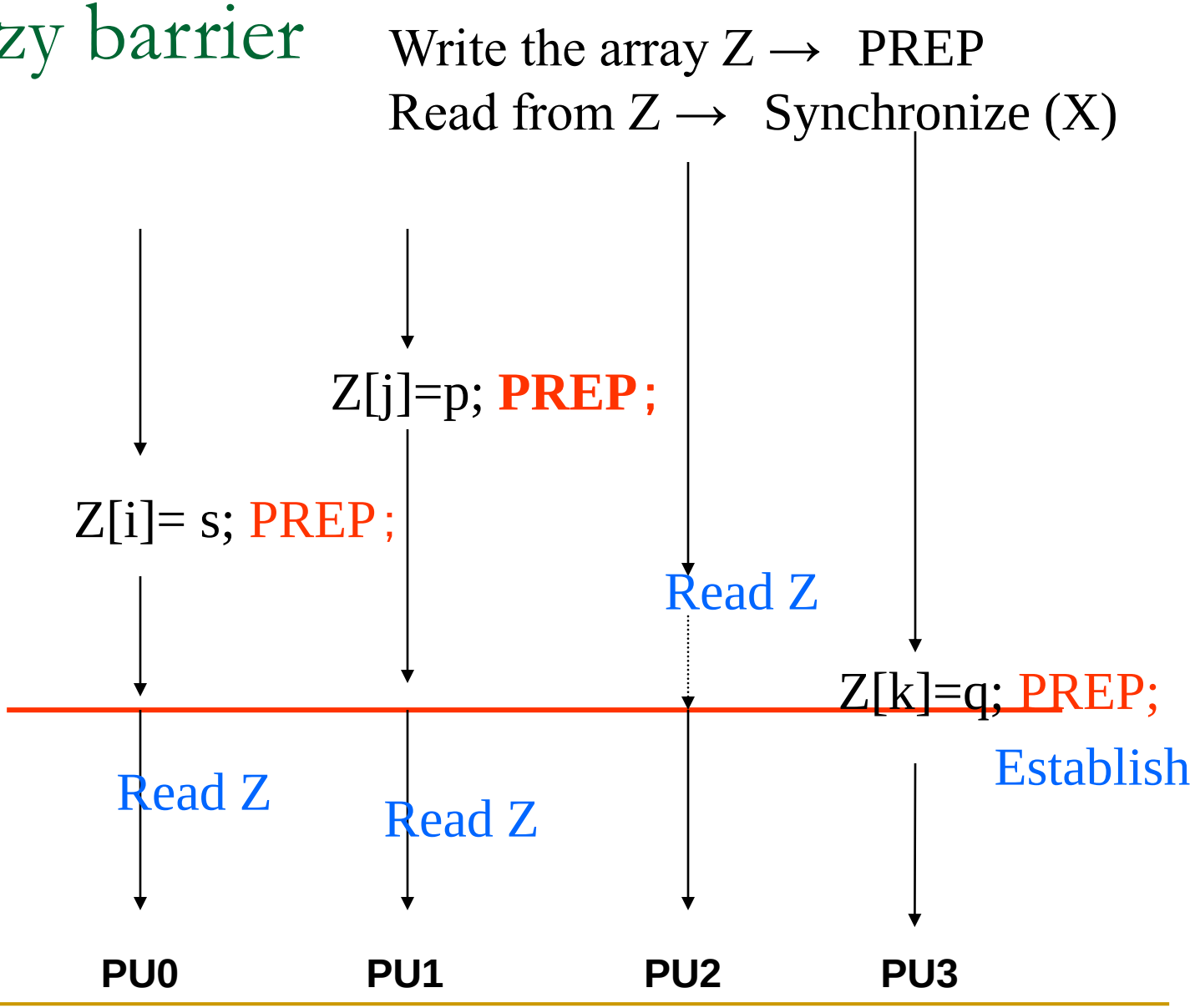
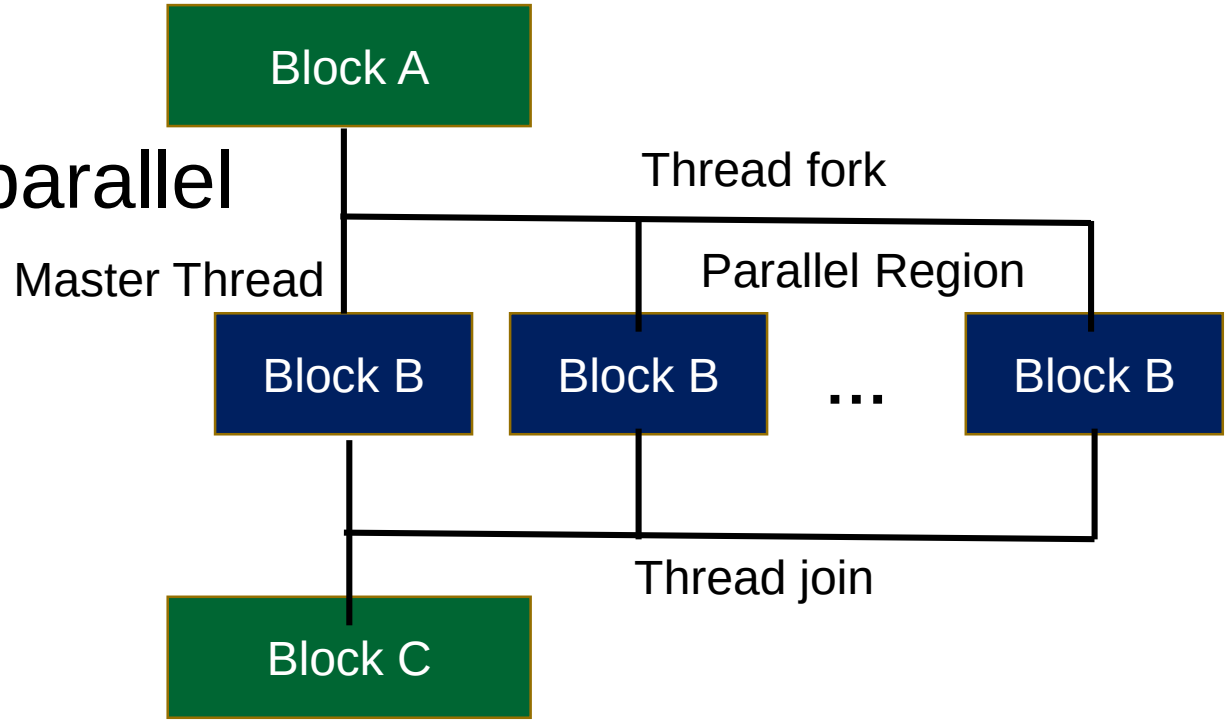


図7　OpenMPの実行モデル

```
Block A
#pragma omp parallel
{
        Block B
{
Block C
```



環境変数: OMP_NUM_THREADS で実行スレッド数を設定

図8　for 文

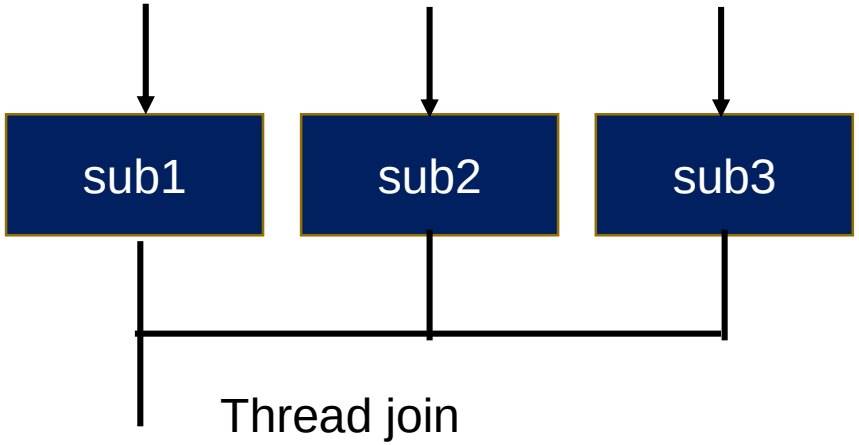
反復は各スレッドに等分に割り当てられる

```
# pragma omp parallel
{
#pragma omp for
        for(i=0; i<1000; i++) {
                c[i]=a[i]+b[i];
        }
}
```

```
# pragma omp parallel for
        for(i=0; i<1000; i++) {
                c[i]=a[i]+b[i];
        }
```

図9　sections 文

```
#pragma omp parallel sections
{
#pragma omp section
    sub1();
#pragma omp section
    sub2();
#pragma omp section
    sub3();
}
```



三つの違った処理が並列に実行され、
終了時に同期される

図10　private sub-directive

```
# pragma omp parallel for private(c)
        for(i=0; i<1000; i++) {
                d[i]=a[i]+c*b[i];
        }
```

c は各スレッドにコピーされる →　高速実行が可能

図11　private sub-directiveの利用

```
# pragma omp parallel for private(j)
        for(i=0; i<100; i++) {
                for(j=0; j<100; j++)
                        a[i]=a[i]+amat[i][j]*b[j];
        }
```

この文をprivate(j)なしに実行したらどうなるだろう？→
全てのスレッドでjが更新される→エラー！

図12　reduction sub-directive

```
# pragma omp parallel for reduction(+:ddot)
        for(i=0; i<100; i++) {
            ddot+= a[i]*b[i];
        }
```

リダクション演算は、データを足し込んでいく演算。
良く用いられるが、並列実行は、このsub-directiveを使わないと難しい