Parallel Programming guideline

Franicsco Espinosa

July 23, 2018

Contents

1 BASICS

- Parallelism - Multiple tasks working together to finish a larger problem faster. Seek of efficiency. Increase in problem size. Extensibility (more nodes). Configurability (different nodes)
- Programming in Parallel - Decomposition of work and data. Mapping to architectures
- Applications- climate simulation, streams in ocean, formation of galaxies, wave simulations, earthquakes.
- $speedup = \frac{time\ 1\ proc.}{time\ proc}$
- $efficiency = \frac{speedup\ p\ proc}{p}$
- **Parallelism exists in:**
  - Processor
    - Bit level, eg. addition.
    - Pipelines, functional units, vectorization.
    - Branch-prediction
  - Memory
    - Prefetching
    - Direct memory access
  - Multiple processors (Hierarchical order)
    1. Processor / Socket
    2. Cores (computational units that sit on one proc chip, socket)
    3. Threads

       Example, a computer that has 2 sockets, 6 cores/socket and 2 threads per core, has 2*6*2 = 24 CPUs (No. of parallel threads)

1.1 CLASSIFICATION OF PARALLELISM

- **SIMD** - Single Instruction Multiple Data (Vectorization, only data parallelism, GPUs)
- **MIMD** - Multiple Instruction Multiple Data (Important for parallelism, dominate parallelism)

1.2 SIMD

This is all about vectorization and intrinsics.

1.3 MIMD

Distributed Memory

Related Software - MPI(Message Passing Interface, PVM, Java(TCP/IP protocols). Send - Recieve.

- MPP - Massively Parallel Programming. Many nodes but only one OS. Expensibe Machines.
- NOW
- Cluster - On each node you have own OS (SuperMuc). Take cheap hardware and connect them

Shared Memory

Related Software - Posix, OpenMP, Load-Store

- UMA - Uniform Memory Access (simmetric multiprocess)
  Centralized shared memory arquitecture type
  Accesses to global memory from alL processors. Each of them have same access latency. Load-Store
- NUMA - Non Uniform Memory Acces
  Memory is distributed among nodes. Local accesses much faster than remote accesses. Still communication between processors is possible. The shared memoty is physically distributed to all proc, known as local memories. Savings in components and cooling. Offers the scalability of MPP and easy programming of SMP. Load-Store, Send-Recieve
  - ccNUMA cach coherence (you have caches in the procs, you might have copies of the same address, and if you change in one cache line, all the other locations should change as well.
  - nccNUMA
  - COMA

Distinction between the type depends on the time for memory access

2 Threading

2.1 Concurrency vs Parallelism

- Concurrency - Having multiple threads available to do differnt task (daemon IO), but not at the same time.
- Parallelism - Having multiple threads available to work in a single task at the same time.

2.2 Threads

Independent stream of instructions that can be scheduled to run as such by the operating system.

2.3 POSIX Threads

- Divided in Thread Management, Mutexes and Condition Variables
- Implemented with a pthread.h header and a thread library

Thread programming example

```c
#include<stdio.h>
#include<stdlib.h> //for malloc
#include<pthread.h>

//funct signature and struct declaration
void *print_message_function(void *ptr);

struct pthread_args {
int thread_id;
int in; int out;
};

int main(){

        int i, num_threads = 4;
        //array of threads
        pthread_t *threads = (pthread_t*)malloc(num_threads*sizeof(pthread_t));
        //array of thread arguments
        struct pthread_args* args = (struct pthread_args*)malloc(num_threads*sizeof(struct
    pthread_args));

        for ( i = 0; i < num_threads; i++){
                args[i].thread_id = i;
                args[i].in=i;
                pthread_create(&threads[i], NULL, print_message_function, args + i);
        }

        for (  i = 0; i < num_threads; i++)
                pthread_join(threads[i],NULL); //instead on NULL (void ptr)&out and use it
    as a void ref

        for (  i = 0; i < num_threads; i++)
                printf("Thread id is %d. Threadid +10 is %d. The double of %d is %d\n", i,
                                        args[i].thread_id, args[i].in, args[i].out);

        free(threads); free(args);

        return 0;
}

void *print_message_function(void *args){
        struct pthread_args *arg = (struct pthread_args*) args;
        arg->out = 2 * arg->in;
        arg->thread_id += 10;
        return NULL;
}
```

Syncronization

Syncronization is used for restricting shared data and resources. The methods usually used are **mutual exclusion, conditional syncronization, atomic variables, reduction.**

Mutex programming example

```
#include<stdio.h>
#include<pthread.h>

// compile with -lpthread flag.
#define NUM 50000
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* increment(void *ptr){

    int *i=(int *)ptr;
    for(int j=0; j < NUM; ++j){
     pthread_mutex_lock (& mutex);
      (*i)++;
      pthread_mutex_unlock (& mutex);
    //  printf("Im the trhead\n");
    }
    return NULL;
}

int main(int argc , char ** argv){

    int i=0;
    pthread_t thr;
    pthread_create(&thr, NULL,& increment ,&i);

    for(int j=0; j < NUM; ++j){
      pthread_mutex_lock (& mutex );
      i++;
      pthread_mutex_unlock (& mutex );
    //  printf("Im main\n");
    }

    pthread_join(thr, NULL);
    printf("Value of i = %d\n", i);
```

for compiling and running g++ -pthread name.cpp -o name
The APIS that must be rembembered are:

- #include<pthread.h>
- pthread_t *thread1; array of threads
- pthread_create(&thread[i], NULL, function, args + i) as addresses
- pthread_join(thread, NULL);
- pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
- pthread_mutex_lock (& mutex);
- pthread_mutex_unlock (& mutex);

3 openMP

Stands for open multiprocessing, and is an API used to program multi-threaded, **shared memory** parallelism. It was born as a standarized version for multi-threaded parallel programming, and **is not intended** for distributed memory.

compile with gcc -fopenmp name.c -o name

3.1 Basic pragmas & openMP instructions

Important #pragma omp [...] statments should not be enclosed in scope signs {...}, only the one for the parallel region

- #pragma omp parallel - creates a team of threads. at the begining, before this, we have a team of only the master thread
- #pragma omp parallel num_threads(4) - directly set num_threads
- omp_set_num_threads(num_threads) - set number of threads
- omp_get_thread_num() - get thread id
- omp_get_num_threads() - get total number of available threads
- omp_set_nested(TRUE) - allow nested threads
- omp_get_wtime() - set time counted from the first call of the function. time = 0 for first call.
- #pragma omp master - only exectued by master thread
- #pragma omp single - single thread runs statment
- #pragma omp critical - read next section on "Syncronization"
- OMP LOCKS
  - omp_lock_t lockvar - declare lock variable
  - omp_init_lock(&lockvar) - initialize a lock
  - omp_destroy_lock(&lockvar) - destroy a lock
  - omp_set_lock(&lockvar) - set lock
  - omp_unset_lock(&lockvar) - free lock
  - logicalvar = omp_test_lock(&lockvar) - check lock and possibly set lock. Returns true if lock was set by the executing thread.

3.2 Syncronization

- Barriers - create a barrier and wait for all threads - #pragma omp barrier
- Critical - create a mutex zone, only one thread at a time - #pragma omp critical
- Atomic - Mutual Exclusion but only to memory update location e.g x++ - #pragma omp atomic

3.3 Loop Worksharing

Important Make loop iterations independent

- #pragma omp for - parallelize a loop. (eq. to #pragma omp parallel for)
- Schedulers. These go after the #pragma omp for
  - schedule(static,chunck) - round robin mode
  - schedule(dynamic, chunck) - decide order on runtime
  - schedule(guided, chunck) - grab big chuncks and reduce to chuncks on runtime, dynamic behaviour
  - schedule (auto) - compiler decides
- #pragma omp for collapse(nested_level) - for multiple-nested loops
- #pragma omp for reduction (+/-/*/min/max:var1) - reduction function. Everything is automatically done.

3.4 Data Enviroment

- pragma omp parallel for - It wil declare the iteration private for you, **but it will not do it** for any other iterative variable(e.g. nested loops). A way of being aware of this, use - default(none).
- #pragma omp default(none/shared/private)- a way to declare everything as a certain type. Use default(none) to debug which variable we are missing to debug. Nothing set is delcared as default(shared)
- #pragma omp shared(var1,...) - variable is visible and editable to every thread
- #pragma omp private(var1,...) - a copy of the variable is created for every thread. **variable is undefined** and returns with the out-of-scope-value.
- #pragma omp firstprivate(var1,...) - same as private, but variables defined with out-of-parallel-region-value, returning with same value
- #pragma omp lastprivate(var1,...) - same as private, but variables defined with out-of-parallel-region-value, returning with last calculated value

3.5 Examples

EXAM QUESTION!!!
1 - Calculate numerically the integral of

$$\int_1^0 \frac{4.0}{1+x^2}\,dx = \pi$$

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

#define MAX_THREADS 4

static long num_steps = 100000000;
double step;
int main ()
{
        int i,j;
        double pi, full_sum = 0.0;
        double start_time, run_time;
        double sum[MAX_THREADS];

        step = 1.0/(double) num_steps;
for(j=1;j<=MAX_THREADS ;j++){
   omp_set_num_threads(j);
   full_sum = 0.0;
          start_time = omp_get_wtime();
#pragma omp parallel private(i)
{
          int id = omp_get_thread_num();
          int numthreads = omp_get_num_threads();
          double x;

          double partial_sum = 0;

#pragma omp single
          printf(" num_threads = %d",numthreads);

          for (i=id;i< num_steps; i+=numthreads){
                  x = (i+0.5)*step;
                  partial_sum += + 4.0/(1.0+x*x);
          }
#pragma omp critical
                  full_sum += partial_sum;
}

          pi = step * full_sum;
          run_time = omp_get_wtime() - start_time;
          printf("\n pi is %f in %f seconds %d threds \n ",pi,run_time,j);
}
}
```

3.6 Sections

- Used for performing a set of structured blocks that are executed by single threads of a team
- When program execution reaches a **omp sections** directive, program segments defined by the following **omp section** directive are distributed for parallel execution among available threads.
- This is the syntax

```
#pragma omp sections <{clause , ...}>
{
        #pragma omp section
        <structured block>

        #pragma omp section
        <structured block>
}
```

- same directives as before private, firstprivate, etc
- omp_set_max_active_levels(levels) - set maximum level of threads within a nested function
- omp_set_nested(1) - activate for aplying it in nested sections
- #pragma omp sections - define a zone for creating sections
- #pragma omp section - define a zone for creating the block for a thread to work on

  Example - Binary Tree

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

struct node
{
  struct node *left, *right;
  int key;
  node(int k):key(k){}
};

void process ( struct node *p){
    usleep (1000000);
    #pragma omp critical
    std::cout<<" element with key: "<<p->key<<" is processed "<<std::endl;
}

void traverse (struct node *p)
{
  #pragma omp parallel
  {
    #pragma omp sections
    {

      #pragma omp section
      {
      if (p->left != NULL)
      traverse (p->left);
      }

      #pragma omp section
      {
      if (p->right != NULL)
      traverse (p->right);
      }
    }
  }
  process(p);
}

int main(){

    struct node *tree = new struct node(0);

    tree->left = new struct node(1);
    tree->right = new struct node(2);
    tree->left->left = new struct node(3);
    tree->left->right = new struct node(4);
    tree->right->left = new struct node(5);
    tree->right->right = new struct node(6);

    omp_set_nested (1);
    omp_set_max_active_levels (2);

    traverse (tree);
    return 0;
}


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

    struct node *tree = new struct node (0);
    tree ->left = new struct node (1);
    tree ->right = new struct node (2);
    tree ->left ->left = new struct node (3);
    tree ->left ->right = new struct node (4);
    tree ->right ->left = new struct node (5);
    tree ->right ->right = new struct node (6);
    omp_set_nested (1);
    omp_set_max_active_levels (2);

    traverse (tree);
    return 0;
}
```

3.7 Tasks

- useful when dealing with unkown loop length, unknown number of parallel sections, or with parallelization of recursive algorithms
- the way it works, is like in a parallel loop, just in this case, a thread creates a task, and when threads are available they will execute those lines of code.
- if the function is recursive, then the previous thread will create another task, and when threads are available, they will perform it.

Example - Traversing a tree

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

void traverse_tasks(tree *node){
        if(node != NULL){

                #pragma omp task
                        traverse_tasks(node->right);

                #pragma omp task //could be removed for optimization
                        traverse_tasks(node->left);

                node->IQ = compute_IQ(node->data);
                genius[node->id] = node->IQ;

                }
}

void traverse(tree *node, int numThreads){
                #pragma omp parallel num_threads(numThreads)
                {
                        #pragma omp single
                                traverse_tasks(node);
                }
}
```

Quick quizz - What is the problem with this code?

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

int main(){
  int id;
  #pragma omp parallel num_threads(4)
  {
    id = omp_get_thread_num();
    #pragma omp critical
    std::cout << "My id is: " << id << std::endl;
  }
}
```

Answer Instead of writing every thread id, we are using the id as shared and is writing the last id that changed the id. The solution is to make it private(id)

4 C++ Multi-Threading

4.1 Basic routines

- #include <threads> #include <mutex> in case of using mutex
- thread_kernel(type &var1, type var2, ...) - define thread function
- std::thread t(&thread_kernel, std::ref(var1), type var2, ...) - create the thread
- t.join() - wait for the thread to finish
- t.detach() - dont wait for thread to finish work. continue master thread work. it may or not finish work.
- for multiple threads, treat them as vectors with the following routines
  - std::vector::<std::thread> Threads - Create vector of threads
  - Threads.push_back(std::thread(kernel, std::ref(var1), var2, ...)) - add thread at the end of the Threads vector
  - Threads[i].join - as with POSIX, join them iteratively

Main difference with the POSIX routine, is that in c++ the kernel and arguments are already included when creating a thread, whereas in POSIX you need a structure to pass the arguments

Example

```
using namespace boost::gil;

template <typename Out>
struct halfdiff_cast_channels;

template <typename SrcView, typename DstView>
void kernel (const SrcView &src, const DstView &dst, int startY, int endY){

typedef typename channel_type<DstView>::type dst_channel_t;

  for (int y = startY; y < endY; ++y)
  {
      typename SrcView::x_iterator src_it = src.row_begin(y);
      typename DstView::x_iterator dst_it = dst.row_begin(y);

      for (int x = 1; x < src.width() - 1; ++x)
      {
          static_transform(src_it[x - 1], src_it[x + 1], dst_it[x],
                           halfdiff_cast_channels<dst_channel_t>());
      }
  }

}

template <typename SrcView, typename DstView>
void x_gradient(const SrcView &src, const DstView &dst, int num_threads) {

 std::vector<std::thread> Threads;
 int startY;
 int endY;

 for (int i = 0; i < num_threads; i++) {

         startY=(src.height()*i)/num_threads;
         endY=(src.height()*(i+1))/num_threads;
         Threads.push_back(std::thread(kernel<SrcView,DstView>,std::ref(src),std::ref(dst),
    startY, endY));
  }

        for (int i=0; i<Threads.size(); ++i)
    {
        Threads[i].join();
    }
}
```

4.2 Syncronization

Example

```
#include <iostream>
#include <thread>
#include <mutex>
#include <list>

std :: list<int> myList; // a global variable
std :: mutex myMutex; // a global mutex

void addToList (int start){
        std::lock_guard <std::mutex > guard( myMutex );//locks m automatically and releases
    it on destruction
        for (int i = start; i < start + 10; i++) {
                myList.push_back(i);
                std::this_thread::sleep_for(std::chrono::milliseconds(2)); //trigger race
    condition
        }
}

int main(){

        std::thread t1(addToList, 0);
        std::thread t2(addToList, 10);
        t1.join ();
        t2.join ();

        for (auto& item : myList)
                std::cout << item << ",";

        std::cout<<"\n";

}
```

Output ./mutexes 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19, **OR**
./mutexes 10,11,12,13, 14,15,16,17,18,19,0,1,2,3,4,5,6,7,8,9,

5 Data Dependencies

5.1 Types of Data Dependencies

- True/Flow Dependence -Read x(S2) After Write x(S1)
  - S1 δ S2
  - S1: x = ...
    S2: ...= x
- Anti Dependence -Write x(S2) After Read x(S1)
  - S1 δ^{-1} S2
  - S1: ... = x
    S2: x = ...
- Outpud Dependence -Write x(S2) After Write x(S1)
  - S1 δ^{o} S2
  - S1: x = ...
    S2: x = ...

5.2 Types of Loop Dependencies

- Loop Independet Dependencies - All dependecies are with iterations. No dependencies across iterations. Iterations can be executed in paralle

  Example
  ```
  for (i = 0; i < 4; i++)
      S1: b[i] = 8;
      S2: a[i] = b[i] + 10;
  ```
- Loop Carried Dependencies - Dependencies across iterations. Iterations **can not** be directly/easily executed in parallel

  Example
  ```
  for (i = 0; i < 4; i++)
      S1: b[i] = 8;
      S2: a[i] = b[i-1] + 10;
  ```

5.3 Loop transformation for parallelism

A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.

A little bit of terminology

- Loop nesting level - Number of surroinding level + 1
- Iteration number - Equal to the value of the iterative variable
- Iteration vector - Given by $\vec{i} := (i_1, i_2, ..., i_n)$ where $i_k, (1 \leq k \leq n)$ represents the iteration number for the loop at nesting level k.
- Iteration space - Set of all possible iteration vector

  Example
  ```
      for (i = 1; i < 3; i++) {
  ```

```
        for (j = 1; j < 4; j++) {
            S: ...
        }
    }
```

Iteration space $\vec{i} := \{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)\}$

- Distance vector - distance of loop carried/independent dependence. Given by $d(\vec{i},\vec{j})_k = j_k - i_k$.
- Direction vector - take the following description when comparing the vector space for the iteration of two statements

$$D(\vec{i},\vec{j})_k :\ = \begin{cases} "<", & d(i,j)_k > 0 \\ "=", & d(i,j)_k = 0 \\ ">", & d(i,j)_k < 0 \end{cases}$$

Example

```
for (i = 1; i < N; i++) {
    for (j = 1; j < M; j++) {
        for (k = 1; k < L; k++) {
            S: A(i + i, j - 1, k)=A(i, j,k )
        }
    }
}
```

The distance vector for this example is $S[(2,2,2)]\ \delta\ S[(3,1,2)]: (1,-1,0)$
The direction vector for this example is $S[(2,2,2)]\ \delta\ S[(3,1,2)]: (<,>,=)$

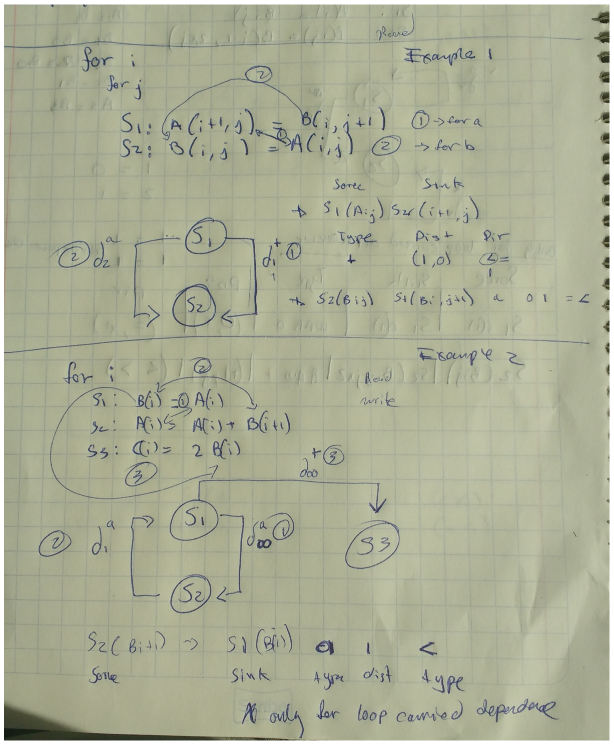
Analizing loops

Rules for building the diagrams and the tables:

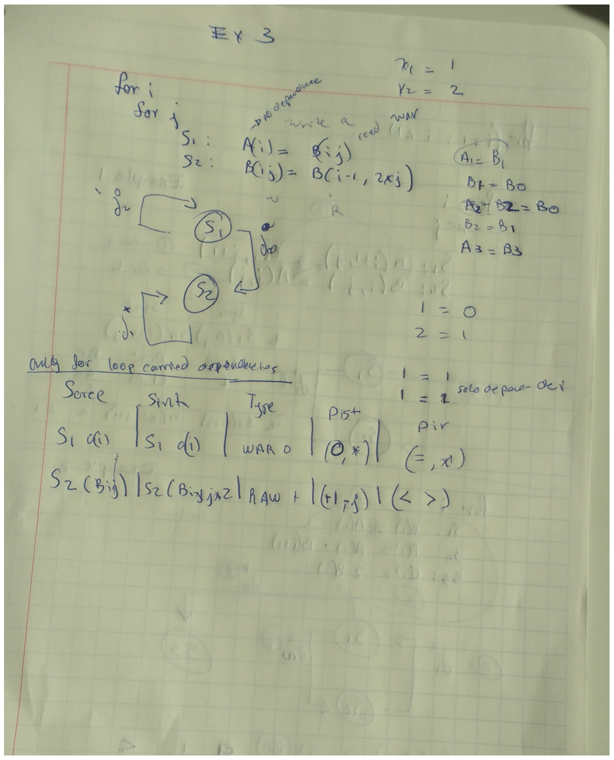
- For the diagram, represent the statement dependance of the lowest level.
- For the table, only add those dependencies that are Loop Carried Dependat

Some Examples

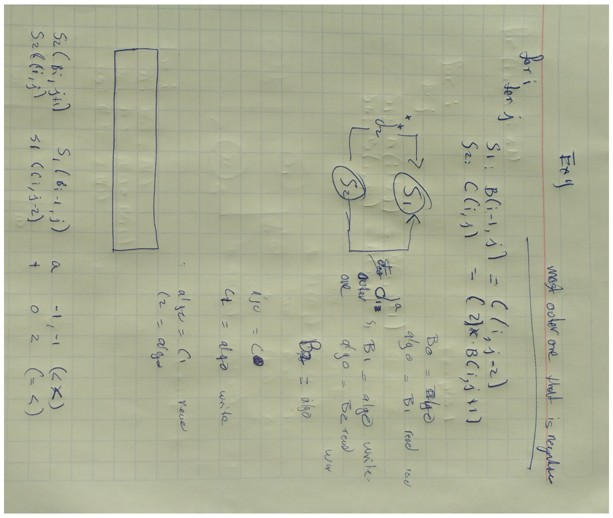
Example 1 and 2



Example 3


Ex 3
for i
for j
S1: A(i) = B(i j)
S2: B(i j) = B(i-1, 2*j)
no dependence
write a read WAR
$x_1 = 1$
$x_2 = 2$
only for loop carried dependencies
Source Sink Type Dist Dir
S1 A(i) S1 A(i) WAR o (0, *) (=, *)
S2 (B i j) S2 (B i, j*2) RAW + (+1, -j) (< >)

Example 4


Ex 4
most outer one that is negative
for i
for j
S1: B(i-1, j) = C(i, j-2)
S2: C(i, j) = (2)* B(i, j+1)
B0 = algo
algo = B1 read
S1 B1 = algo write
algo = B2 read
B2 = algo
S1
S2
d2
d1
outer one
C0 = algo
C1 = algo write
algo = C1 read
C2 = algo
S2(Bi, j+1) S1(Bi-1, j) a -1, -1 (<X)
S2 (Ci, j) S1 (Ci, j-2) + 0 2 (=<)

Analizing transformations

1. Loop Interchange - The interchange is valid up to the leftmost non-"=" that is not a "<" in the direction vector. The interchange means that insted of: **do i -> do j**, it is valid: **do j -> do i**
2. Loop Distribution / Loop Fission - Transforms loop-carried dependences into loop-independent dependences. Safety of loop distribution:
   - Two sets of statements in a loop nest can be distributed into separate loop nests, if no dependence cycle exists between those groups.
   - The order of the new loops has to preserve the dependences among the statement sets.
   - Example

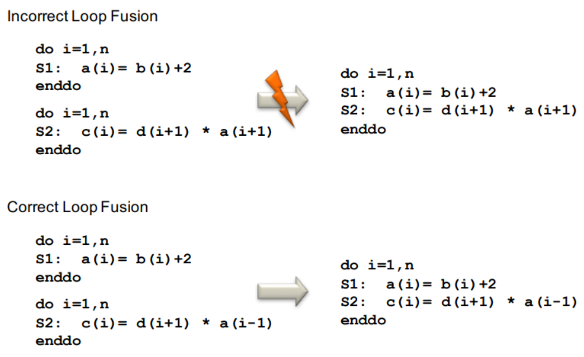
```
do j=2,n
S1:  a(j)= b(j)+2
S2:  c(j)= a(j-1) * 2
enddo
```

```
do j=2,n
S1:  a(j)= b(j)+2
enddo

do j=2,n
S2:  c(j)= a(j-1) * 2
enddo
```

3. Loop Fusion

- Is the dual transformation. Increases granularity
- Might reduce parallelism
- example



Incorrect Loop Fusion

```
do i=1,n
S1:  a(i)= b(i)+2
enddo

do i=1,n
S2:  c(i)= d(i+1) * a(i+1)
enddo
```

```
do i=1,n
S1:  a(i)= b(i)+2
S2:  c(i)= d(i+1) * a(i+1)
enddo
```

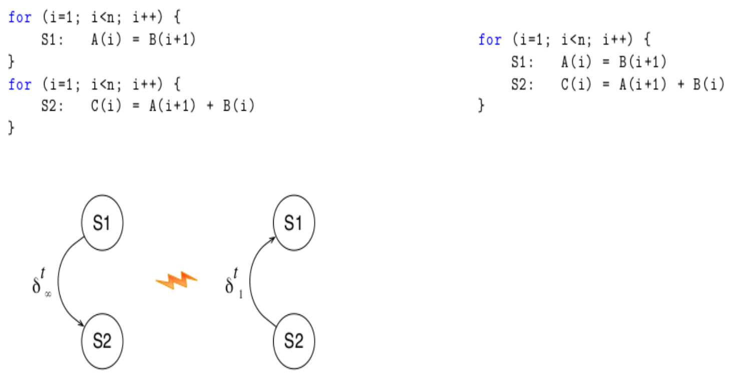
Correct Loop Fusion

```
do i=1,n
S1:  a(i)= b(i)+2
enddo

do i=1,n
S2:  c(i)= d(i+1) * a(i-1)
enddo
```

```
do i=1,n
S1:  a(i)= b(i)+2
S2:  c(i)= d(i+1) * a(i-1)
enddo
```

- example(conflict)



```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i+1) + B(i)
}
```

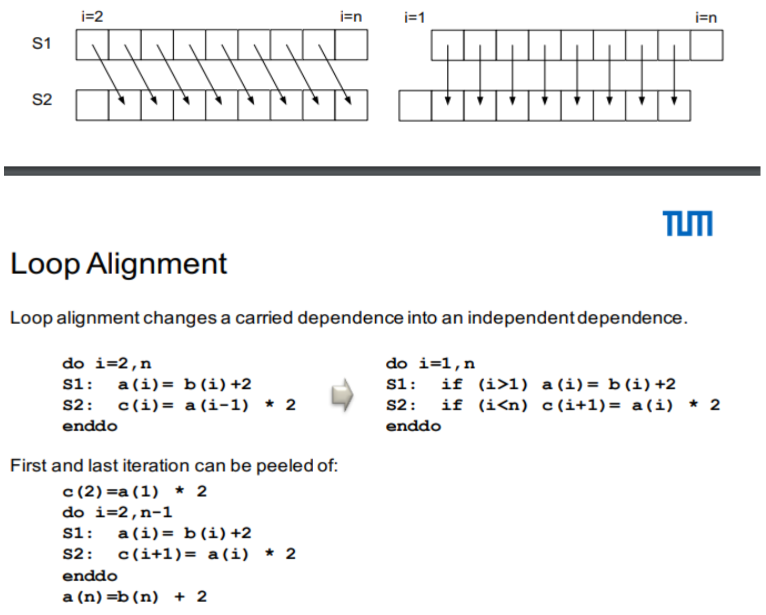
```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
    S2:   C(i) = A(i+1) + B(i)
}
```

Figure 1: dependencies are not conserved

4. Loop Alignment

- Changes a carried dependence into an independent dependence.
- We can align a loop when it doesnt contain no recurrence (dependence cycle) **and** the distance of each dependence is a constant independent of the loop index

- example



Loop Alignment

Loop alignment changes a carried dependence into an independent dependence.

```
do i=2,n
S1:  a(i)= b(i)+2
S2:  c(i)= a(i-1) * 2
enddo
```

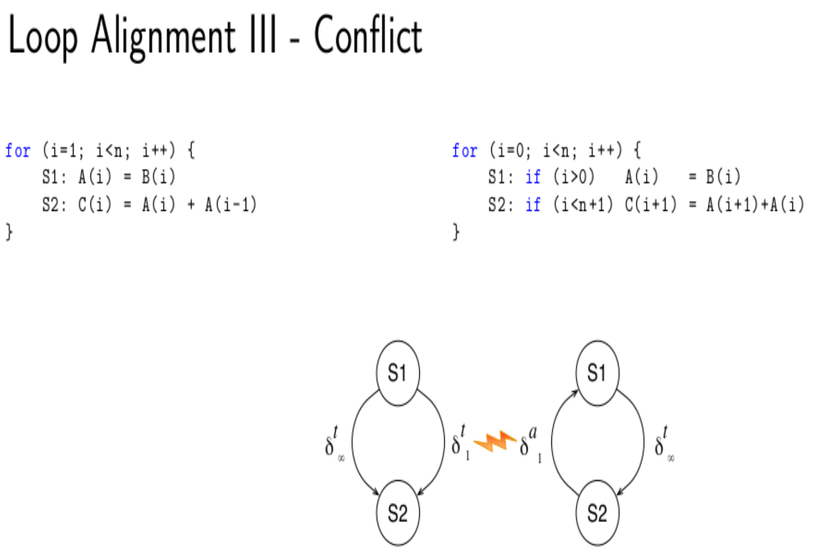
```
do i=1,n
S1:  if (i>1) a(i)= b(i)+2
S2:  if (i<n) c(i+1)= a(i) * 2
enddo
```

First and last iteration can be peeled of:

```
c(2)=a(1) * 2
do i=2,n-1
S1:  a(i)= b(i)+2
S2:  c(i+1)= a(i) * 2
enddo
a(n)=b(n) + 2
```

Figure 2: example

- example(conflict)



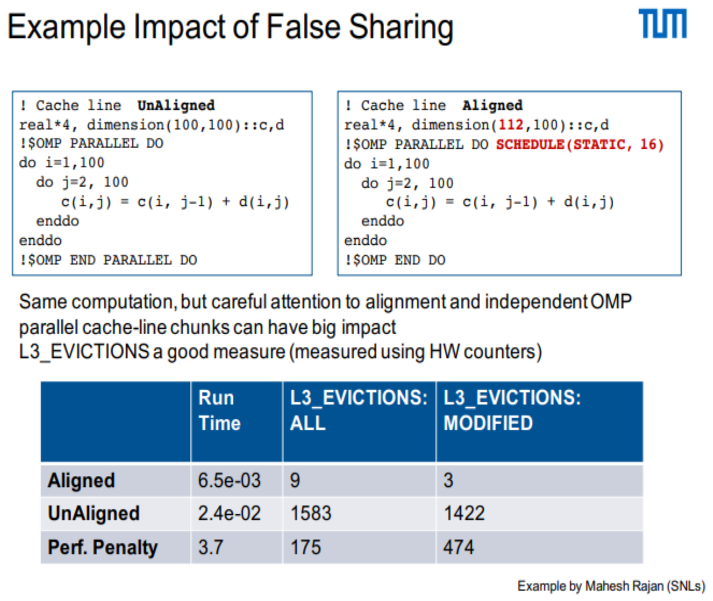
Loop Alignment III - Conflict

```
for (i=1; i<n; i++) {
    S1: A(i) = B(i)
    S2: C(i) = A(i) + A(i-1)
}
```

```
for (i=0; i<n; i++) {
    S1: if (i>0)   A(i)   = B(i)
    S2: if (i<n+1) C(i+1) = A(i+1)+A(i)
}
```

Figure 3: Conflict dependencies are not conserved

5.4 False Sharing



Example Impact of False Sharing

```
! Cache line  UnAligned
real*4, dimension(100,100)::c,d
!$OMP PARALLEL DO
do i=1,100
  do j=2, 100
     c(i,j) = c(i, j-1) + d(i,j)
  enddo
enddo
!$OMP END PARALLEL DO
```

```
! Cache line  Aligned
real*4, dimension(112,100)::c,d
!$OMP PARALLEL DO SCHEDULE(STATIC, 16)
do i=1,100
  do j=2, 100
     c(i,j) = c(i, j-1) + d(i,j)
  enddo
enddo
!$OMP END DO
```

Same computation, but careful attention to alignment and independent OMP parallel cache-line chunks can have big impact
L3_EVICTIONS a good measure (measured using HW counters)

	Run Time	L3_EVICTIONS: ALL	L3_EVICTIONS: MODIFIED
Aligned	6.5e-03	9	3
UnAligned	2.4e-02	1583	1422
Perf. Penalty	3.7	175	474

Example by Mahesh Rajan (SNLs)

Avoid False Sharing by:

- add some padding sum[NUM_THREADS][cache_block_size / sizeof(type)]
- using private variables

6 Distributed Memory, MPI