Parallel Programming guideline

Franicsco Espinosa

July 24, 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
- Coherency and Consistency
  - Coherency - Reasoning about updates to one memory location. Loads/Stores are preserved. All stores become visible. All processors see the same order of writes.
  - Consistency - Reasoning about updates to several memory locations. "A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the programâĂİ

Distinction between the type depends on the time for memory access

2 Threading

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.

Threads & POSIX Threads

A Thread is an independent stream of instructions that can be scheduled to run as such by the operating system.

- 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 [1] 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);

[1] Application Programming Interface - subroutines that offers a library for different softwares

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
- Flush - Synchronizes data of the executing thread with main memory, e.g., copies in register cache - #pragma omp flush [(list_of_variables)]

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

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

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  #define MAX_THREADS 4
5
6  static long num_steps = 100000000;
7  double step;
8  int main ()
9  {
10            int i,j;
11            double pi, full_sum = 0.0;
12            double start_time, run_time;
13            double sum[MAX_THREADS];
14
15            step = 1.0/(double) num_steps;
```

```
16
17
18 for(j=1;j<=MAX_THREADS ;j++){
19    omp_set_num_threads(j);
20    full_sum = 0.0;
21             start_time = omp_get_wtime();
22 #pragma omp parallel private(i)
23 {
24             int id = omp_get_thread_num();
25             int numthreads = omp_get_num_threads();
26             double x;
27
28             double partial_sum = 0;
29
30 #pragma omp single
31             printf(" num_threads = %d",numthreads);
32
33             for (i=id;i< num_steps; i+=numthreads){
34                     x = (i+0.5)*step;
35                     partial_sum += + 4.0/(1.0+x*x);
36             }
37 #pragma omp critical
38                     full_sum += partial_sum;
39 }
40
41             pi = step * full_sum;
42             run_time = omp_get_wtime() - start_time;
43             printf("\n pi is %f in %f seconds %d threds \n ",pi,run_time,j);
44 }
45 }
```

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);
```

```
69
70      omp_set_nested (1);
71      omp_set_max_active_levels (2);
72
73      traverse (tree);
74      return 0;
75  }
```

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

```
1  #include "familytree.h"
2  #include <omp.h>
3
4  void traverse_tasks(tree *node){
5          if(node != NULL){
6
7                  #pragma omp task //node will be firstprivate by default
8                          traverse_tasks(node->right);
9
10                 #pragma omp task //could be removed for optimization
11                         traverse_tasks(node->left);
12
13                 node->IQ = compute_IQ(node->data);
14                 genius[node->id] = node->IQ;
15
16                 }
17 }
18
19 void traverse(tree *node, int numThreads){
20                 #pragma omp parallel num_threads(numThreads) //important
21                 {
22                         #pragma omp single //important
23                                 traverse_tasks(node);
24                 }
25 }
```

Quick quizz - What is the problem with this code?

```
1  #include <iostream>
2  #include <omp.h>
3
4  int main(){
5    int id;
6    #pragma omp parallel num_threads(4)
7    {
8      id = omp_get_thread_num();
9      #pragma omp critical
10     std::cout << "My id is: " << id << std::endl;
11   }
12 }
```

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

```
1  using namespace boost::gil;
2
3  template <typename Out>
4  struct halfdiff_cast_channels;
5
6  template <typename SrcView, typename DstView>
7  void kernel (const SrcView &src, const DstView &dst, int startY, int endY){
8
9  typedef typename channel_type<DstView>::type dst_channel_t;
10
11   for (int y = startY; y < endY; ++y)
12   {
13       typename SrcView::x_iterator src_it = src.row_begin(y);
14       typename DstView::x_iterator dst_it = dst.row_begin(y);
15
16       for (int x = 1; x < src.width() - 1; ++x)
17       {
18           static_transform(src_it[x - 1], src_it[x + 1], dst_it[x],
19                            halfdiff_cast_channels<dst_channel_t>());
20       }
21   }
22
23 }
24
25 template <typename SrcView, typename DstView>
26 void x_gradient(const SrcView &src, const DstView &dst, int num_threads) {
27
28  std::vector<std::thread> Threads;
29  int startY;
30  int endY;
31
32  for (int i = 0; i < num_threads; i++) {
33
34          startY=(src.height()*i)/num_threads;
35          endY=(src.height()*(i+1))/num_threads;
36          Threads.push_back(std::thread(kernel<SrcView,DstView>,std::ref(src),std::ref(dst),
     startY, endY));
37   }
38
39         for (int i=0; i<Threads.size(); ++i)
```

```
40      {
41          Threads[i].join();
42      }
43  }
```

4.2 Syncronization

Example

```
1   #include <iostream>
2   #include <thread>
3   #include <mutex>
4   #include <list>
5
6   std :: list<int> myList; // a global variable
7   std :: mutex myMutex; // a global mutex
8
9   void addToList (int start){
10          std::lock_guard <std::mutex > guard( myMutex );//locks m automatically and releases
        it on destruction
11          for (int i = start; i < start + 10; i++) {
12                  myList.push_back(i);
13                  std::this_thread::sleep_for(std::chrono::milliseconds(2)); //trigger race
        condition
14          }
15  }
16
17  int main(){
18
19          std::thread t1(addToList, 0);
20          std::thread t2(addToList, 10);
21          t1.join ();
22          t2.join ();
23
24          for (auto& item : myList)
25                  std::cout << item << ",";
26
27          std::cout<<"\n";
28
29  }
```

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 $S[(2,2,2)]\ \delta\ S[(3,1,2)]$: $(<,>,=)$

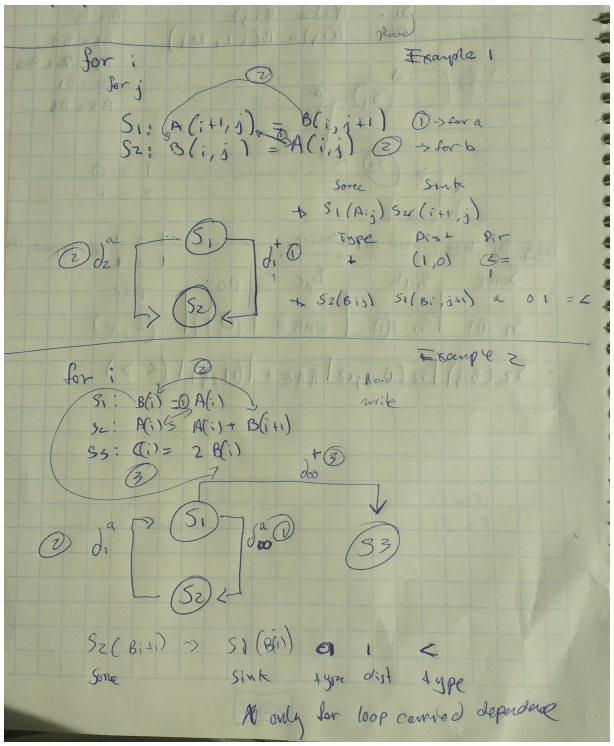
Analizing loops

Rules for building the diagrams and the tables:

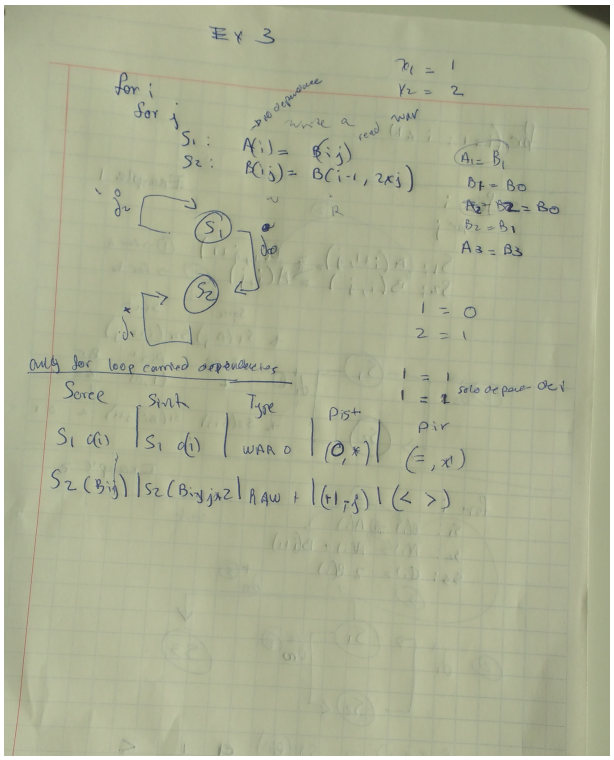
- For the diagram, represent the statement dependance of the lowest level.
- For the table, only add those dependencies that are Loop Carried Dependat

Some Examples

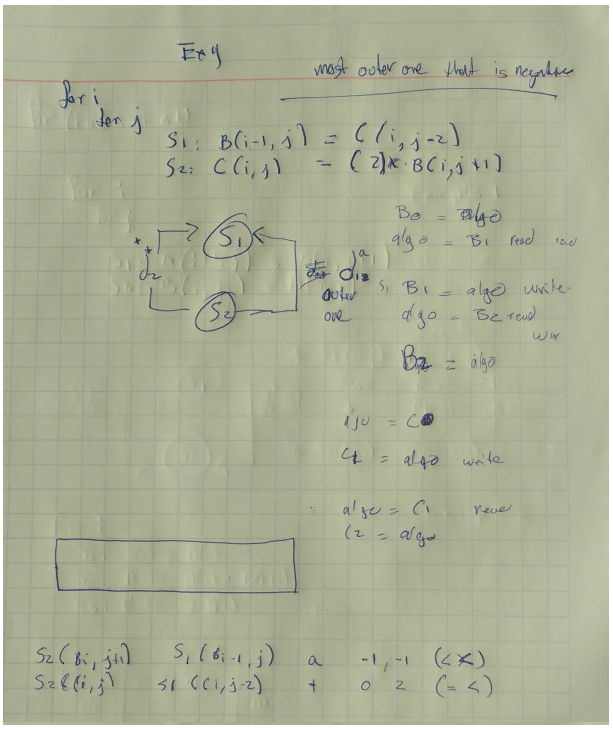
Example 1 and 2



Example 3



Example 4



Analizing transformations

1. Loop Interchange - The interchange is valid when all the direction vector is "=" **or** 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

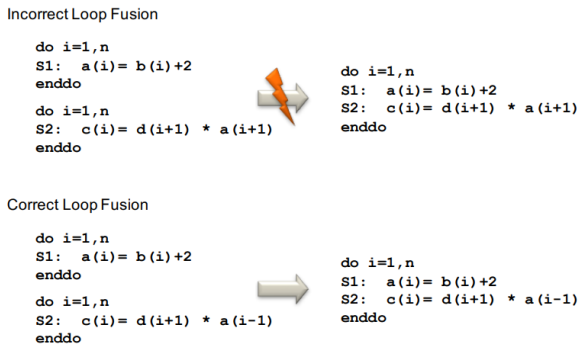
```
do j=2,n
S1:  a(j)= b(j)+2
S2:  c(j)= a(j-1) * 2
enddo
```

```
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
- For having an equivalent loop, the dependencies type, direction **and the dependency level** must be conserved, t
- example



- example(conflict)

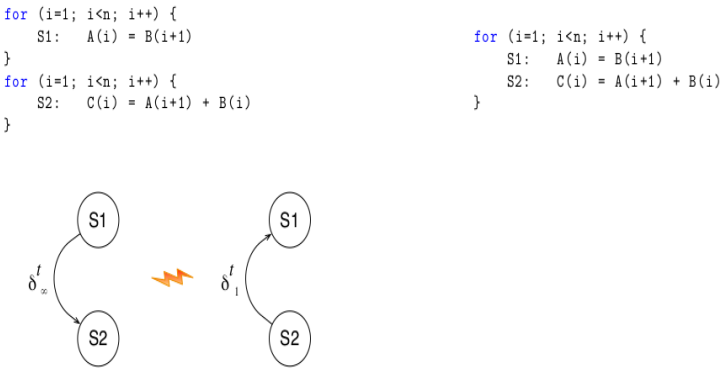


Figure 1: dependencies are not conserved

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

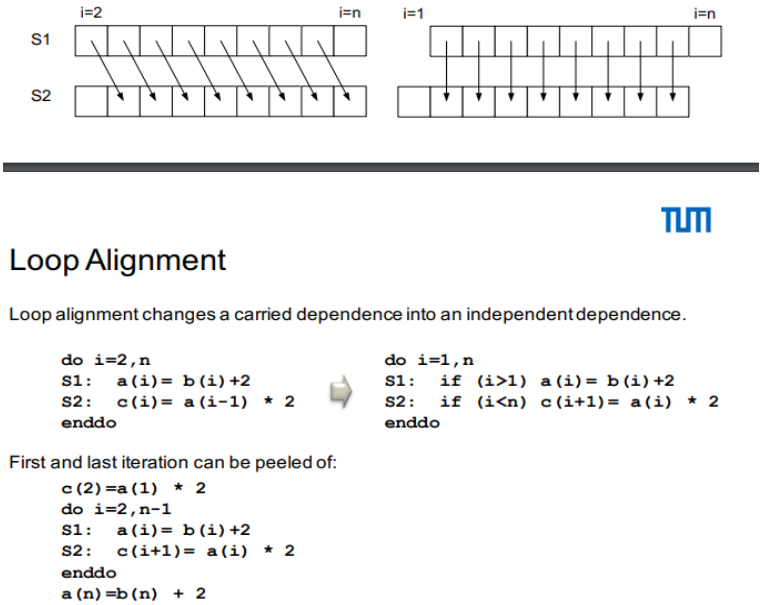


Figure 2: example

- example(conflict)

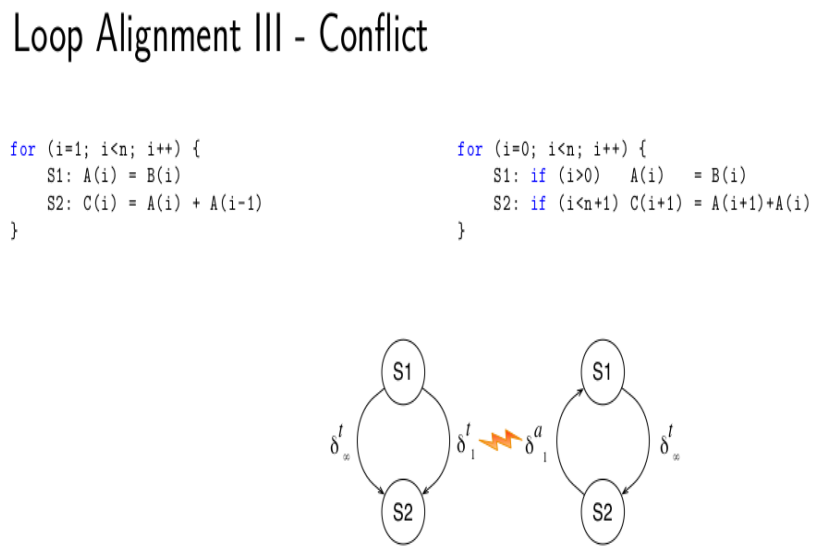
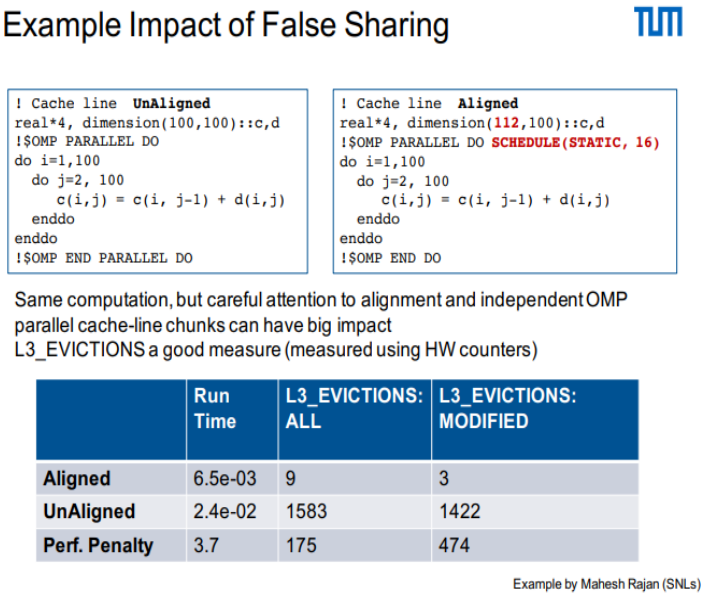


Figure 3: Conflict dependencies are not conserved

5.4 False Sharing



Avoid False Sharing by:

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

6 Distributed Memory, MPI

6.1 Remarks

- Compute Chip -> Computre Card -> Node Card -> Cabinet -> System
- Std Protocol TCP/IP, Ethernet
- cooling reduces overhead
- controlled accesses
- Applications
  - Weather forecast
  - Nuclear Physics
  - Oil & Gas reservoir modeling
  - Genomics
  - Material Science
  - CFD & Crashes
  - Astrophysics
  - 6 pillars - National Security, Energy Security, Economic Security, Scientifc Discovery, Earth Systems, Health Care
- HPC uses distributed memory - Multiple users ->jobs scheduled by management system ->free resources take the job

6.2 MPI

6.2.1 Basis

- send and recieve operations
- support C and Fortran
- implemented as a library
- API specified
- no runtime defined interactions
- not a protocol

6.2.2 Principles

- independent processes with own code - different(independent) data type representation
- Common usage : Single Program Multiple Data (its not such a model, but most commonly used as such)
- debbuging with printf()
- Check tutorial slides session_8 for installation
- compile with mpicc code.c -o code / run with mpirun -np 4 ./code

6.3 Coding

6.3.1 6 Basic Commands

- #include<mpi.h> - library call
- **MPI_Init**(&argc, &argv) / MPI_Finalize() - start and end mpi
- **MPI_COMM_WORLD** - Communicators
  - Argument to many functions
  - Addresing is relative to a communicatior
  - Ranks go from 0 to N-1
  - A process can have different ranks in different communicators
  - Communication across communicators anre not possible
  - MPI_COMM_SELF - refer only to the communicator of the process calling this function, in contrast to MPI_COMM_WORLD
- **MPI_Comm_size**(MPI_COMM comm, &size) - number of processors. Every code should check the size of MPI_COMM_WORLD
- **MPI_Comm_rank**(MPI_COMM comm, &rank) - rank of processor, relative to communicator
- **int MPI_Send**(void buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm) - Send one message to another MPI process on a given communicator.
  - instructions
    - buf - Address of the send buffer
    - count - Number of dasta elemts of type dtype to be sent

        * dtype - Data type (SHORT INT LONG lONG_LONG UNSIGNED_SHORT UNSIGNED UNSIGNED_LONG FLOAT SOUBLE C_COMPLEX C_DOUBLE_COMPLEX BYTE PACKED
        * dest - Reciever (rank of tharger MPI process in comm)
        * tag - Message tag
        * comm - Communicator
    - Initiates message send and blocks execution until send buffer can be reused. Deadlocks

- **int MPI_Recv**(void buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status) - Recieves of a message from another MPI process on a given communicator
    - instructions
        * buf - Address of the reecieve buffer
        * count - Number of dasta elemts of type dtype to be sent
        * dtype - Data type
        * source - Reciever (rank of tharger MPI process in comm)
        * tag - Message tag
        * comm - Communicator
    - Blocks execution until matching message has been recieved. Deadlocks
    - MPI_ANY_SOURCE & MPI_ANY_TAG instead of *source* and *tag*.
    - message buffer must be at least as long as the message, partial recieves are permited.

6.4 Fancy commands

-

6.4.1 More commands

- **MPI_Wtime**() - Return wall clock time since a reference time stamp in the past.
- **MPI_Status** - In C is a structure that includes the following fields
    - instructions
        * MPI_SOURCE sender of the message
        * MPI_TAG - message tag
        * MPI_ERROR - error code
    - MPI_Get_Count(const MPI_Status *status, MPI_Datatype dtype, int *count) - Actual lenghtof the recevied message

Basic Example

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main (int argc, char** argv){
  double value;
  int size, rank;
  MPI_Status s;
  MPI_Init (&argc, &argv);

  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  value=MPI_Wtime();
  printf("MPI Process %d of %d (value=%f)\n", rank, size, value);

  if (rank>0) //proc:0 does not send to proc:-1 (does not exist)
```

```
16      MPI_Recv(&value, 1, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, &s); //proc send to the left
17
18    if (rank<size-1)// last proc does not recieve from right (no one sent anything to him)
19      MPI_Send(&value, 1, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD); //proc recieves from right
20
21    if (rank==size-1)
22      printf("Value from MPI Process 0: %f\n",value);
23
24    MPI_Finalize ();
25  }
```

6.5 Create a Logical Topology

- Define base elements
- Define neighborhood information
- Establish neighborhood direction
- Understand frequency of communication
- Example - Halo exchange(domain subdivision)

6.6 Blocking vs Non Blocking Communication

6.6.1 Blocking

Blocking communication[2] consists of four send modes and one receive mode. The four send modes are:

- Standard (MPI_Send) - The sending process returns when the system can buffer the message or when the message is received and the buffer is ready for reuse.
- Buffered (MPI_Bsend) - The sending process returns when the message is buffered in an application-supplied buffer. Avoid using the MPI_Bsend mode. It forces an additional copy operation.
- Synchronous (MPI_Ssend) - The sending process returns only if a matching receive is posted and the receiving process has started to receive the message.
- Ready (MPI_Rsend) - The message is sent as soon as possible.

Invoke any mode by using the correct routine name and passing the argument list. Arguments are the same for all modes.

6.6.2 Non Blocking

MPI provides nonblocking counterparts for each of the four blocking send routines and for the receive routine. e.g. MPI_Isend, MPI_Ibsend, MPI_Issend. Nonblocking calls have the same arguments, with the same meaning as their blocking counterparts, plus an additional argument for a request.

MPI_Isend(void *buf, int count, MPI_datatype dtype, intdest, int tag, MPI_Comm comm, MPI_Request *req)

About **req**, it specifies the request used by a completion routine when called by the application to complete the send operation. To complete nonblocking sends and receives, it is possible to use:

[2] https://www.ibm.com/support/knowledgecenter/en/SSF4ZA_9.1.4/pmpi_guide/msg_send_receive.html

- **MPI_Wait**(MPI_Request *request, MPI_Status *status)

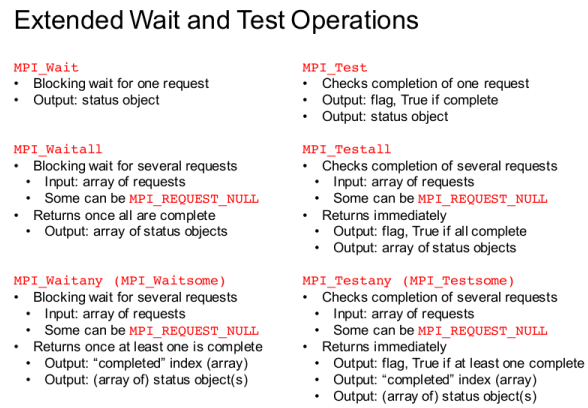
```
{
        MPI_Request req;
        MPI_Status status;
        int msg[10];
        ...
        MPI_Irecv(msg, 10, MPI_INT, MPI_ANY_SOURCE, 42,
        MPI_COMM_WORLD, &req);
        ...
        <do work>
        ...
        MPI_Wait(&req, &status);
        printf(âĂİProcessing message from %i\nâĂIJ,
        status.MPI_SOURCE);
        ...
}
```

- **MPI_Test**(MPI_Request *request, int *flag, MPI_Status *status)

```
{
        MPI_Request req;
        MPI_Status status;
        int msg[10], flag;
        ...
        MPI_Irecv(msg, 10, MPI_INT, MPI_ANY_SOURCE, 42,
        MPI_COMM_WORLD, &req);
        do
        {
        ...
        <do work>
        ...
        MPI_Test(&req, &flag, &status);
        } while (flag==0);
        printf(âĂİProcessing message from %i\nâĂIJ,
        status.MPI_SOURCE);
}
```

The completion of a send indicates that the sending process is free to access the send buffer. **The completion of a receive** indicates that the receive buffer contains the message, the receiving process is free to access it, and the status object that returns information about the received message, is set.

Extended Wait & Test Operations



6.6.3 Global Coordination - Useful commands

- Synchronization
  - Barrier - **MPI_Barrier**(MPI_Comm comm) - Sycnronizes all MPI Proc, i.e., no process can return from the call until every other proc have called it.
- Communication
  - Broadcast - **MPI_Bcast**(void *buf, int count, MPI_Datatype dtype, int root, MPI_Comm comm)

    The contents of the send buffer is copied to all other MPI processes. No tags. no. snd elments must be no. recv lements
  - Gather - **MPI_Gather**(void *sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

    Get data from all MPI processes into one local array in *root*
  - Scatter - **MPI_Scatter**(*same args*)

    Opposite of MPI_Gather. Buffer from *root* scatters its data to every proc.
  - There are more Collectives. See lecture slides 08-MPIPart2
- Reduction
  - Global value returned to one or all proc.
  - Combination with subsequent scatter
  - Parallel prefix operations
  - Commands
    - MPI_Reduce(void* sbuf, void* rbuf, int count, MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm)
    - MPI_Allreduce(*same args exept for root*)

7 Hybrid - MPI and Threads

- initialize as MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
  FUNNELED - only main thread makes calls to mpi

8 Intrinsics SIMD

- vectorization Example

```
1  #include"dgemm.h"
2  #include <immintrin.h>
3  #include <stdio.h>
4
5  void dgemm(float *a, float *b, float *c, int n)
6  {
7  __m256 va, vb, vtemp, vsum;
8  int count= 0;
9  float  sum = 0;
10 float temp[8];
11
12     for(int i = 0; i < n; i++){
13         for(int j = 0; j < n; j++){
14             vsum = _mm256_setzero_ps();
15             for(int k = 0; k < n; k+=8){
16               va = _mm256_loadu_ps(&a[i * n + k]);
17               vb = _mm256_loadu_ps(&b[j * n + k]);
18               vtemp = _mm256_mul_ps(va,vb);
```

```
19                  vsum = _mm256_add_ps(vtemp,vsum);
20                  }
21
22                _mm256_storeu_ps(temp, vsum);
23                sum += temp[0] + temp[1] + temp[2] + temp[3] + temp[4] + temp[5] + temp[6]
      + temp[7];
24                c[i * n + j]=sum;
25                sum=0;
26            }
27      }
28  }
```

9 Sequential Code Optimization

- Loop Unrolling
- Loop Invariant - avoid executing invariants many times
- Inlining
- Manual and Automatic Vectorization
- Flag Optimization
  - -O0 - no optimization
  - -O1 - optimize
    Better register allocation, dead code elimination
  - -O2 - optimize even more
    More aggressive CSE, remove redundant instructions
  - -O3 - optimize yet more
    Aggressive inlining, vectorization
  - Os - optimize for size
  - -Og - optimize debugging experience
  - -Ofast -disregard strict standards compliance.
  - Floating - point optimizations
    -march=native (in addition) - architecture tuning