IN4200 Exam Sheet

IN4200 Students
(Dated: May 2022)

CONTENTS

I. INTRODUCTION

Link to edit here and exam questions from inf3380 here.

II. CODE OPTIMIZATION (SERIAL)

A. General guidelines to optimize code

1. Common sense (minimize unnecessary data-traffic and floating point operations)

   For periodic functions such as sin() and cos() it is sometimes sufficient only to calculate parts of it.

   If you use the same value(s) several times, precalculate it outside of loop.

2. Reuse data from cache as much as possible.

   Go through matrix row-wise (in C).

   Fusion, unroll, blocking.

3. Avoid branching.

4. Reduce number of complex instructions (exponent, sin, cos, pow) to avoid pipeline bubbles.

   Multiplication is better than division.

5. Reduce overhead

   If you have nested for loops, have the smaller one outside.

6. Consider code balance versus machine balance to see where the bottleneck is (data traffic or FLOPS).

7. Use compiler flags, for instance -O3 for vectorization and other black magic.

B. Fusion

Fuse two loop together. Advantages

1. Reduce possible unnecessary loading of elements used in both loops.

2. Reduce looping overhead

Disadvantages

1. Limited register resources. If the code body of each iteration is to large one might get so-called register spilling which can lead to performance degradation.

2. If one loop contains advanced mathematical calculations, fusion could hinder SIMD and/or pipelining

C. Loop unrolling

Explicitly write out iterations of the loop and go through the loop in a strided fashion.This transformation can be undertaken manually by the programmer or by an optimizing compiler.

Advantages:

1. For 2D loops (or more) we can reuse elements from register and thus reducing data traffic.

2. Reduction in loop overhead. The overhead in "tight" loops often consists of instructions to increment a pointer or index to the next element in an array (pointer arithmetic), as well as "end of loop" tests. However, this is arguable not a big factor in the total time use.

Disadvantages

1. Limited register resources (same as fusion).

2. Less readable

Example:

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      C[i] += A[i][j]*B[j];
    }
}
```

```
for (i=0; i<N; i+=m) {
    for (j=0; j<N; j++) {
      C[i+0] += A[i+0][j]*B[j];
      C[i+1] += A[i+1][j]*B[j];
      // ...
      C[i+m-1] += A[i+m-1][j]*B[j];
    }
}
// remainder code in case (N%m)>0 ....
```

Here we make use B[j] from register. The combination of loop-unrolling and fusion is often called "unroll and jam".

D. Blocking

Blocking extends the concept of loop unrolling to take into account the cache use in the other dimension. This is useful if you load values both from arrays row-wise and column-wise in the same iteration. Consider for instance the matrix tranpose $A = B^T$

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] = B[j][i];
    }
}
```

Even with loop unrolling applied to the j-loop, a small cache size will lead to a great deal of wasted loads to cache regarding B[j][i]. Thus we can block the i-loop, meaning that we divide it into chunks. Blocking both loops we can even iterate the matrices in chunks of size b*b such that loop blocking + unrolling looks like this

```
for (ii=0; ii<N; ii+=b) {
  istart = ii; istop = ii+b-1;
  for (jj=0; jj<N; jj+=b) {
    jstart = jj; jstop = jj+b-1;

    for (i=istart; i<=istop; i+=m)
      for (j=jstart; j<=jstop; j++) {
        A[i+0][j] = B[[j]i+0];
        A[i+1][j] = B[j][i+1];
        // ....
        A[i+m-1][j] = B[j][j+m-1];
      }
  }
}
// remainder code in case (N%b)>0
// and/or (N%m)>0 ...
```

In addition, if we choose the blocking factor b to match a multiple of cache line lengths the use of cache will be highly optimized.

III. FORMULAS

A. Data traffic: Latency and bandwidth

A typical way of modelling the time for data transport is

$$T = T_l + \frac{m}{B}, \tag{1}$$

where T_l is latency (s), m is message in bytes and B is bandwith (bytes/s)

$$\text{Code balance:} \quad B_c = \frac{\#\text{ data traffic (store + load)}}{\#\text{ FP operations}}$$

$$\text{Machine balance:} \quad B_m = \frac{\text{memory bandwidth}}{\text{peak performace [FP/s]}},$$

If B_c is larger than B_m, the speed of the computation is determined by the memory traffic, otherwise the speed of the computation is determined by the number of FP operations.

$$\begin{aligned}\frac{B_c}{B_m} &= \frac{\text{data traffic}}{\text{memory bandwidth}} \left(\frac{\text{FP operations}}{\text{peak performace}}\right)^{-1} \\ &= \frac{\text{Time for memory traffic}}{\text{Time for FP operations}}\end{aligned}$$

B. Division of work and load imbalance

One solution to define start and stop values in loop-division

$$\begin{aligned}\text{my\_start} &= (rank * N)//p \\ \text{my\_stop} &= ([rank + 1] * N)//p \\ \text{my\_len} &= \text{my\_stop} - \text{my\_start}\end{aligned}$$

Another solution, where we just define the length, is given as

$$\text{my\_len} = N//\text{nr\_procs} + \text{my\_rank} < N\%\text{nr\_procs}$$

Notice that the two above approaches divides the work differently and should not be mixed.

If the work assigned to p processors is denoted respectively by $W_1, W_2, \ldots, W_p$, we can then define the load imbalance factor as

$$\frac{\max_i (W_i) - \min_i (W_i)}{\min_i (W_i)} \tag{2}$$

C. Speedup for parallel code

We define speedup as

$$\text{speedup} = \frac{\text{serial runtime}}{\text{parallel runtime}} = \frac{\text{parallel performance}}{\text{serial performance}}$$

The overall problem size is normalized as $s+p = 1$ in the following laws, where s is the serial, non-parallelizable fraction, p is the perfectly parallelizable fraction.

1. Amdahls law (strong scaling)

For a fixed-size problem (strong scaling) we get

$$\begin{aligned}T_f^s &= s + p && \text{(serial code)} \\ T_f^p &= s + \frac{p}{N} && \text{(parallel code)}\end{aligned}$$

where T_f^s is the time for serial code and T_f^p for the parallel code using N workers. The speedup S_f by parallizing is then (ideally) given as

$$S_f = \frac{T_f^s}{T_f^p} = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{1-s}{N}} \quad \text{(Amdahl's law)}$$

For $N \to \infty$ the maximum speedup is $1/s$.

2. Gustafson's law (weak scaling)

For weak scaling, the total amount of work is scaled with some power of N on the parallel part

$$s + pN^\alpha, \qquad \alpha > 0$$

such that we get

$$\begin{aligned}T_v^s &= s + pN^\alpha && \text{(serial code)} \\ T_v^p &= s + pN^{\alpha-1} && \text{(multiple workers)}\end{aligned}$$

The most typically choice is $\alpha = 1$. Hence, when assuming $s + p = 1$ (i.e., $s + p$ is normalized to a single time unit), the speedup (Gustafson's law) becomes

$$S_v = \frac{T_v^s}{T_v^p} = \frac{s+pN}{s+p} = \frac{s+(1-s)N}{s+1-s} = s + (1-s)N$$

Here speedup can be arbitrarily large when $N \to \infty$.

IV. PARALLELIZATION

A. Shared memory computers (UMA and ccNUMA architectures)

A shared-memory parallel computer is a system in which a number of CPUs work on a common, shared physical address space. We generally distinguish between two varieties in terms of main memory access

1. Uniform Memory Access (UMA): Latency and bandwidth are the same for all processors and all memory locations
2. cache-coherent Nonuniform Memory Access (ccNUMA): memory is physically distributed but logically shared. Due to the distributed nature, memory access performance varies depending on which CPU accesses which parts of memory ("local" vs. "remote" access).

1. Cache coherence

Cache coherence mechanisms are required in all cache-based multiprocessor systems, whether they are of the UMA or the ccNUMA kind. This is because copies of the same cache line could potentially be in several CPU caches. If one of those gets modified the content of other caches' becomes outdated. Cache coherence protocols ensure a consistent view of memory under all circumstances.

B. First touch

When working with ccNUMA systems, ideally the data is allocated "closest" (with fastest transfer speed) to the CPU that is going to use it. This is handled with the first touch policy for memory pages: A page gets mapped into the locality domain of the processor that first writes to it. Thus one must have in mind:

1. You must use static scheduler and identical chunk-size (Dynamic/guided is decided at runtime).
2. The calloc function will be counterproductive

C. False sharing

False sharing happens when two or more OpenMP threads load the same cache line from memory to their respective private cache, and that the OpenMP threads update elements on different locations on the same cache line. Due to the cache coherence policy, expensive memory traffic has to be carried out and it effectively serializes the updates. A simple example of false sharing is shown in the example below:

```
# assume that 'counter' is an integer array of
  length num_threads
# assume that 'a' is an integer array of
  length n
#pragma omp parallel for
for (i=0; i<n; i++)
    counter[my_rank] += a[i]
```

Can be fixed with an offset for the addressees in the couner array, or simply with a private variable to sum up before a reduciton.

- False sharing is not the same as race condition.
- False sharing is not specific for NUMA, it also happens for UMA

D. OpenMP

See this webpage for documentation.
Clauses:

- schedule(type[,chunk_size]), types are static, dynamic, guided, auto, runtime. (Applies to the for directive.)
- nowait (Overrides the barrier implicit in a directive.)
- private (Specifies that each thread should have its own instance of a variable.)
- firstprivate (Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.)
- lastprivate (Specifies that the enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section (# pragma sections).)
- Reduction(operation:var), operations are +,*,-,^,&,&&,|,||. (Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.)

Directives:

- parallel (Defines a parallel region, which is code that will be executed by multiple threads in parallel.)
- for (Causes the work done in a for loop inside a parallel region to be divided among threads.)
- single (Lets you specify that a section of code should be executed on a single thread, not necessarily the main thread.) Implicit barrier.

- master (Specifies that only the main thread should execute a section of the program.) There is no implied barrier.
- critical (Specifies that code is only executed on one thread at a time.) No barrier
- barrier (Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.)

Functions:

- omp_set_num_threads (Sets the number of threads in upcoming parallel regions.)
- omp_get_num_threads (Returns the number of threads in the parallel region.)
- omp_get_thread_num (Returns the thread number of the thread executing within its thread team.)

E. Distributed memory systems

It is as it sounds: Memory is not shared.

F. OpenMP vs MPI

- MPI will lead to extra memory usage(ghost values, duplicated arrays/variables, MPI internals memory usage)
- MPI requires explicit work/data division, extra programming effort and possibly overhead.
- MPI must handle risk of deadlocks
- MPI gives better control of the code
- Data locality is better with MPI (no race condition, first touch, false sharing) than OpenMP
- Shared cache may benefit OpenMP, not MPI
- OpenMP gives simpler code, easier to read
- MPI scales better. OpenMP cannot handle to many threads, but the number of MPI processors can be arbitrarily large.

G. 1D and 2D Partitioning

In the case of using MPI to perform calculations on a $N \times N$ matrix, we can choose to divide the work using 1D or 2D partitioning. 1D partitioning means dividing the work such that information must only be communicated in one dimension (i.e vertically or horizontally), whereas 2D requires communication in two (i.e vertically and horizontally). Which method of partitioning is ideal, is dependant on several factors. In the case of 1D we send large amount of data for each communication, but fewer communications is required. In the case of 2D we send less amount of data, but require more communication. For a $n \times n$ array we get that data traffic time T goes as (ignoring boundary effects)

$$1\text{D} : T \propto 2(t_s + t_w n)$$
$$2\text{D} : T \propto 4(t_s + t_w \frac{n}{\sqrt{p}})$$

H. Hybrid MPI/OpenMP

Hybrid MPI/OpenMP is the basic idea to allow any MPI process to spawn a team of OpenMP threads. Having one or few MPI processes per node; OpenMP is used to further exploit the parallelism within the node. The main benefits of hybrid MPI/OpenMP

- Re-use of data in shared caches
- Exploiting additional levels of parallelism
- Reducing MPI overhead
- Multiple levels of overhead; Distribute overhead on both MPI and openMP.

There are two basic hybrid programming approaches: **vector mode** and task mode.

1. Vector mode

All MPI functions are called outside OpenMP parallel regions.

- Advantage: Easy and safe to implement. An existing pure MPI code can be turned hybrid just by adding OpenMP worksharing directives in front of the time-consuming loops.
- Disadvantage: This is a "rigid" approach that limits its possibility for hiding MPI overhead.

2. Task mode

Task mode allows handling MPI communication within OpenMP parallel regions.

- Advantage:

  A high level of flexibility

  Better task decomposition and decoupling of communication and computation.

- Disadvantage:

  Harder to implement.

  Can no longer use the convenient OpenMP worksharing parallelization directives (# pragma omp for), and workload has to be distributed "manually" among the threads.

Since MPI may be implemented in environments with poor or no thread support, the MPI standard distinguishes four different levels of thread compatibility:

1. MPI_THREAD_SINGLE: Only one thread will execute (used in the vector mode)
2. MPI_THREAD_FUNNELED: The process may be multithreaded, but only the main thread will make MPI calls.
3. MPI_THREAD_SERIALIZED: The process may be multithreaded, and multiple threads may make MPI calls, but only one at a time; MPI calls are not made concurrently from two distinct threads.
4. MPI_THREAD_MULTIPLE: Multiple threads may call MPI anytime, with no restrictions.

By using: int MPI_Init_thread(int *argc, char ***argv, int required, int *provided);, one can make sure that the thread environment is as intended. Example

```
int main( int argc, char *argv[] )
{
  int errs = 0, provided, claimed;
  MPI_Init_thread( &argc, &argv, MPI_THREAD_MULTIPLE, &provided );
  MPI_Query_thread( &claimed );
  if (claimed != provided) {
    errs++;
    printf("Query thread gave thread level %d but Init_thread gave %d\n",
           claimed, provided );fflush(stdout);
}
  MPI_Finalize();
  return errs;
}
```

V. GENERAL CONCEPTS

A. SIMD instructions

SIMD (Single instruction multiple data) is the opportunity to perform the same operation on multiple data points simultaneously. Typically, the operations in SIMD units include basic arithmetic operations (such as addition, subtraction, multiplication, negation) and other operations such as absolute (abs) and square root (sqrt) on "wide" registers (typical size 16-32 bytes) each holding multiple numerical values. This is equivalent to vectorization. This is useful when working with arrays in for-loop. (See this page.)

B. Prefetcching

Prefetching supplies the cache with data ahead such that it is already loaded when needed. When successful, this hide the latency of memory load. Prefetching is not manually controlled by the programmer but takes places when the hardware pre-fetcher can detect regular access patterns in the code.

C. Pipelining

Pipelining is an example of instruction level parallelism. Different units of the CPU is in charge of different operations such as addition and multiplication. Pipelining is sometimes compared to a manufacturing assembly line in which different parts of a product are being assembled at the same time although ultimately there may be some parts that have to be assembled before others are. However, when doing the same operations repeatedly in a loop, one unit can go to the next iteration when finished with its part of the calculation and thus drastically increase the number of instructions executed per clock cycle. "Throughput" is the mena number of operations which is done per clock cycle.

For a pipeline of depth m, executing N independent operations takes $N + m - 1$ cycles. The speedup versus "no pipeling" is

$$\frac{T_{\text{seq}}}{T_{\text{pipe}}} = \frac{N \cdot m}{N + m - 1}$$

The throughput, average number of operations finished per cycle, can be calculated as

$$\frac{N}{T_{\text{pipe}}} = \frac{1}{1 + \frac{m-1}{N}}$$

Pipeline bubbles is when one operation takes a lot more time than other, and thus this is a limiting factor.

D. Cache

Caches are low-capacity, high-speed memories that are commonly integrated on the CPU die. There exist different levels of cache, often L1 (typical size 32-64KB per core), L2 and L3, where the lowest numbers is fastest but with lowest capacity. When loading data, it needs to be loaded from memory (RAM) all the way to the lowest level of cache and then enter the register. However, if the data is reused in the near future it will still be in the cache, which make the next fetching very fast.

Suppose accessing a data item in cache is a factor of τ faster than accessing the main memory. Let β denote the cache reuse ratio. Suppose access time to main memory is denoted by T_{m}, access time to cache is thus $T_{\text{c}} = T_{\text{m}}/\tau$. The average access time will be

$$T_{\text{av}} = \beta T_{\text{c}} + (1 - \beta) T_{\text{m}}$$

Performance gain due to cache can be calculated by

$$G(\tau, \beta) = \frac{T_{\text{m}}}{T_{\text{av}}} = \frac{\tau T_{\text{c}}}{\beta T_{\text{c}} + (1 - \beta)\tau T_{\text{c}}} = \frac{\tau}{\beta + (1 - \beta)\tau}$$

1. *Cache lines (or cache entries, cache blocks)*

When fetching data from memory it is always done in multiples of cache lines. The cache line is generally 16 to 256 bytes (2 to 32) words / double precision values.

E. FMA (Fused multiply-add)

FMA (Fused multiply-add) is a technical term representing the floating-point operation performing multiplication and addition in one clock cycle. This is relevant if you want the computer to achieve maximal performance.

F. Dead code

Some compilers may recognize code which results is never used later on, and thus removing it from the executable program. This is something to keep in mind when planing timing of "useless" code. Can be safely avoided by just printing the results of the algorithm by the end.

G. Types and sizes

TABLE I. Integer types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

TABLE II. Floating-point types

Type	Storage size	Value range	Precision
float	4 bytes	1.2E-38 to 3.4E+38	6 decimal places
double	8 bytes	2.2E-308 to 1.8E+308	15 decimal places
long double	16 bytes	3.4E-4932 to 1.2E+4932	18 decimal places

Note: The actual values can be machine-dependent.

H. stack and heap

Stack is used for static memory allocation and Heap for dynamic memory allocation, both stored in the computer's RAM.

$$\text{int u[10]; } \rightarrow \text{ Stack} \tag{3}$$

$$\text{int } *\text{u} = \text{malloc}(10*\text{sizeof(int)}); \rightarrow \text{ Heap} \tag{4}$$

You can use the stack if you know exactly how much data you need to allocate before compile time and it is not too big. You can use heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data. Beware that main function and other functions and declarations are also stored in the stack. Stack is faster, however there is a limit on how much you can store on the stack. There is also a no risk of memory fragmentation when using stack, however there is that risk when using heap.

I. Race condition

Race conditions: when multiple threads tries to change the same values in memory at the same time. For instance when doing += on the same element.

J. Contention

If two processors try to access the same place in memory simultaneously the processors will fight for the bandwidth.

VI. MORE STUFF

A. Handy math

$\boxed{ABA}$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

B. Format for sparse matrix

For at $N \times N$ matrix with N_{nz} non-zero values. Three 1D arrays:

- *Val* with length N_{nz}, containing the non-zero values
- *Col_idx* with length N_{nz}, with the original column position of the non-zero values.
- *row_ptr* with length $N + 1$, containing the indices for which new row start in the array val

C. Indexing for multidim arrays with 1D underlying structure

$$\text{2D: } \text{index}(i, j) = i * n_y + j$$
$$\text{3D: } \text{index}(i, j, k) = i * n_y * n_z + j * n_z + k$$

D. 2D array (M x N) with underlying 1D structure

Approved by Andreas $\boxed{ABA}$

```
int M, N;
int **arr;

arr = (int **)malloc(M * sizeof *arr);
arr[0] = (int *)malloc(M * N * sizeof **arr);
for (size_t i = 0; i < M; i++)
{
    arr[i] = &arr[0][N * i];
}
```

E. Buzzwords to remember

1. Race conditions (when multiple threads tries to write/store at the same time
2. Bottleneck
3. Parallelism (The ability of a code to be paralyzed)