High Performance Computing
Unit - 2

Scalar Profiling :-

Profiling is a process of gathering data about program's use of resource.
Profiling helps to determine which portions of program have greatest potential for reducing overall run-time through code optimization.

A common profiling strategy is to find out how much time is spent in different functions, and may be even lines of code in order to identify hotspots.

In many situation we might identify Hot Spots but can not find the actual reason for a performance bottleneck (i.e we dont know starting point of problem).
Sometimes delay is caused by data access.

We can overcome such issues by hardware performance counters.
(bottleneck, Stalling)

Hardware Performance Counters :-

Mostly all of the modern processor provides this. It can count the no. of occurences of various events :-

- → Bus transactions (cache line transfer)
- → Loads and stores
- → floating point operation
- → mispredicted branches
- → pipeline stalls
- → instruction executed

Function & Line Based Runtime Profiling :-

In general 2 techniques comes under function and line based profiling :-

1) Code instrumentation → It is the process of preparing to collect data. Basically it involves solution of the method by which code will be collected as well as process of collecting those codes. It is more accurate.

2) Sampling → In this we takes samples from line of codes and does the profiling.

Sampling	Code Instrumentation
→ Less disruptive	→ More disruptive
→ Less accurate	→ More accurate

One of the most widely used tool for function profiling is "GPROF".

"gprof" provides data on number of calls and time spent in each function in a program. It uses both instrumentation and sampling to collect a flat function profile as well as a callgraph profile known as butterfly graph.

Example :-

% time	Cumulative seconds	self seconds	calls	self ms/call	total ms/call
70.45	5.14	5.14	26074562	0.00	0.00
26.01	7.03	1.90	4000000	0.00	0.00
3.72	7.30	0.27	100	2.71	73.03

↳ flat profile

→ % time → % of runtime used by this function
→ Commulative seconds → Commulative sum of runtime of all function up to and including this one.
→ self seconds → No. of seconds used by this function.
→ calls → No. of time it is called.
→ self ms/call → Average no. of milliseconds per call.
→ total ms/call → Average no. of ms per call that were spend on this including its callees.

Flat profile already have lot of information, but it does not reveal how runtime contribution of a cutain function is composed of several different callers, which other function is called from it, and which contribution to run time they in turn incur. It is provided by Callgraph profile.

Simple measures, large impact :-

While writing the code, we need to consider simple measures we can take that will create a large impact on performance.

Some of the following measures are :-

(i) Elimination of common subexpression :-

It is an optimization that is often considered as task for compilers.
Basically we try to save time by precalculating parts of complex expression and assigning them to temporary ~~values~~ variables before code construct starts.

In case of loops, this optimization is also known as loop invariant code motion.

// inefficient :-

```
do i=1, N
  A(i) = A(i) + S+r*sin(n)
end do
```

⇒

// Efficient :-

```
temp = S+r*sin(n)
do
  A(i) = A(i) + tmp
enddo.
```

(ii) Avoiding Branches :-

"Tight" loops (i.e loops that have few operation in them) are typical candidates for software pipelining, loop unrolling and other optimization techniques. Compiler optimization fails if the loop body ~~condi~~ contains conditional branches.

// inefficient

```
do j=1,N
  do i=1,N
    if(i.ge.j) then
      sign=1.d0
    else if (i.lt.j) then
      sign=-1.d0
    else
      sign=0.d0
    endif
    C(j)=C(j)+sign*A(i,j)*B(i)
  enddo
enddo
```

⇒

// efficient

```
do j=1,N
  do i=j+1,N
    C(j)=C(j)+A(i,j)*B(i)
  enddo
enddo
do j=1,N
  do i=1,j-1
    C(j)=C(j)-A(i,j)*B(i
  enddo
enddo
```

(iii) Using SIMD instruction sets :-

The use of SIMD in microprocessor is often termed "vectorization". Generally speaking, a "vectorizable" loop in this context will run faster if more operation can be performed with single instruction.

The Role of Compilers :-

Most high-performance codes benifit from employing compiler-based optimizations. Every modern compiler has command line switches that allows fine grained tuning of available optimization options.

Advantages of using compilers :-
→ faster execution
→ less memory
→ better performance

Some of the compiler based optimization techniques to enhance performance :-

(i) General Optimization options :-
All compilers have their own standard optimization options (-O0, -O1, - -).
At higher levels, optimizing compilers mix up source lines, detect and eliminate
(a) "Redundant" variables.
(b) Rearrange arithmatic expression

(ii) Inlining :- It is one of the optimization in compiler which tries to save overhead by inserting the complete code of a function or subroutine at the place where it is called.

(iii) Register optimization :- It is one of the most vital but also most complex tasks of the compiler to care about register usage. The compiler tries to put operands that are used "most often" into register and keep them there as long as posible, given that it is safe to do so.
Inling will helps with register optimization.

(iv) Using Compiler logs :-
We can check history of compiler logs to identify already existing instruction and since it already have such thing in the past, so all data must be available. So it will take less time and hence throughput will increase.

(v) Aliasing :- An alias occurs when different variables point directly or indirectly to a single area of storage. Aliasing refers to assumption made during optimization about which variable can point to or occupy the same storage area.

(vi) Computational Accuracy :- (Write in your own words)

C++ Optimization

Optimization is a coding activity. In traditional software development processes, optimization takes place after code Complete, during integration and testing phase of a project, when the performance of whole program can be observed.

~~C++ is the com~~

One of the noradicable illusion about C++ is that the compiler should be able to see through all abstractions and C++ program has. First and foremost C++ should be seen as a language that enable complexity management.

C++ Optimization Dynamic memory Management

Dynamic memory allocation refers to allocating system memory at run time.

C++ codes have frequent allocation and deallocation of memories. So, we need to manage these allocation and deallocation. This is where dynamic memory management comes into the picture.

Strategies :-

(i) Lazy construction (ii) Static Construction

(i) Lazy construction :-

Lazy construction of an object means that its creation is defered / delayed until it is first used. It is used to improve performance by avoiding wasteful computation and reduce program memory requirement.

(Note → Iska example me, we can give example of e-comerce website, where when user does not want to invoice of the order then why to generate that.)

(ii) Static Construction :- Moving the construction of an object to the outside of a loop or block or making it static altogether may even be faster than lazy construction if the object is used often. It is called automatically when the object is initialized.

Static object can't be created again. There is no need to declare again.

C++ Optimizations Temporaries :-

C++ fosters an "implicit" programming style where automatic mechanisms hide complexity from the programer. A frequent problem occurs with expression containing chains of overloaded operators.

As an example, consider this vec3d class which represents a vector in 3-d space.

```
class vec3d {
    double x, y, z;
    friend vec3d operator *(double, const vec3d&);
    public:
vec3d(double _x=0, double y=0, double _z=0) {
        x = -x;
        y = -y;
        z = -z;
}
vec3d(const vec3d &other);
vec3d & operator = (const vec3d &other);
vec3d operator + (const vec3d &other);
Vec3d temp;
temp.x = x + other.x;
temp.y = y + other.y;
temp.z = z + other.z;
}
};
```

```
vec3d operator*(double s, const vec3d& v){
    vec3d tmp(s*v.x, s*v.y, s*v.z);
}

vec3d a, b(2,2), c(3);
double x=1.0, y=2.0

a = x*b + y*c;
```

In the above example,

1. Constructor for a, b, c, d is called.
2. operator *(x, b) is called.
3. vec3d constructor is called to initalize tmp in operator *(double --).
4. Since tmp is destroyed once it returns to vec3d copy constructor is invoked to make a temporary copy of result.

⋮ and so on with + operator overloading.

* Although compiler may eliminate the local tmp object by so-called return value optimization. using required implicit temporary directly instead of tmp, it is striking how much code gets executed for this strategy.

A straightforward optimization at cost of readability is to use compound assignment operator like +=, *=, etc.

e.g. a = y*c;
a+= x*b;

C++ Optimizations :- Loop Iterators & Kernels

The runtime of scientific application tend to be dominated by loops or loops nests, and the compiler's ability to optimize those loops is pivotal for getting good code performance.

Operator overloading, as convenient as it may be, hinders good loop optimization. Performance get degraded.

Iterators :- iterations also plays an role in optimization

Eg.
```
template <class T> T sprod(const vector<T> &a, const
                           vector<T> &b) {
    T result = T(0);
    int s = a.size();
    for(int i=0; i<s; ++)
        result += a[i]*b[i]
    return result;
}
```

In line 7, const T & vector<T>::operator[] is called twice to obtain the current entries from a and b.

Data Accen Optimization (DAO) :-

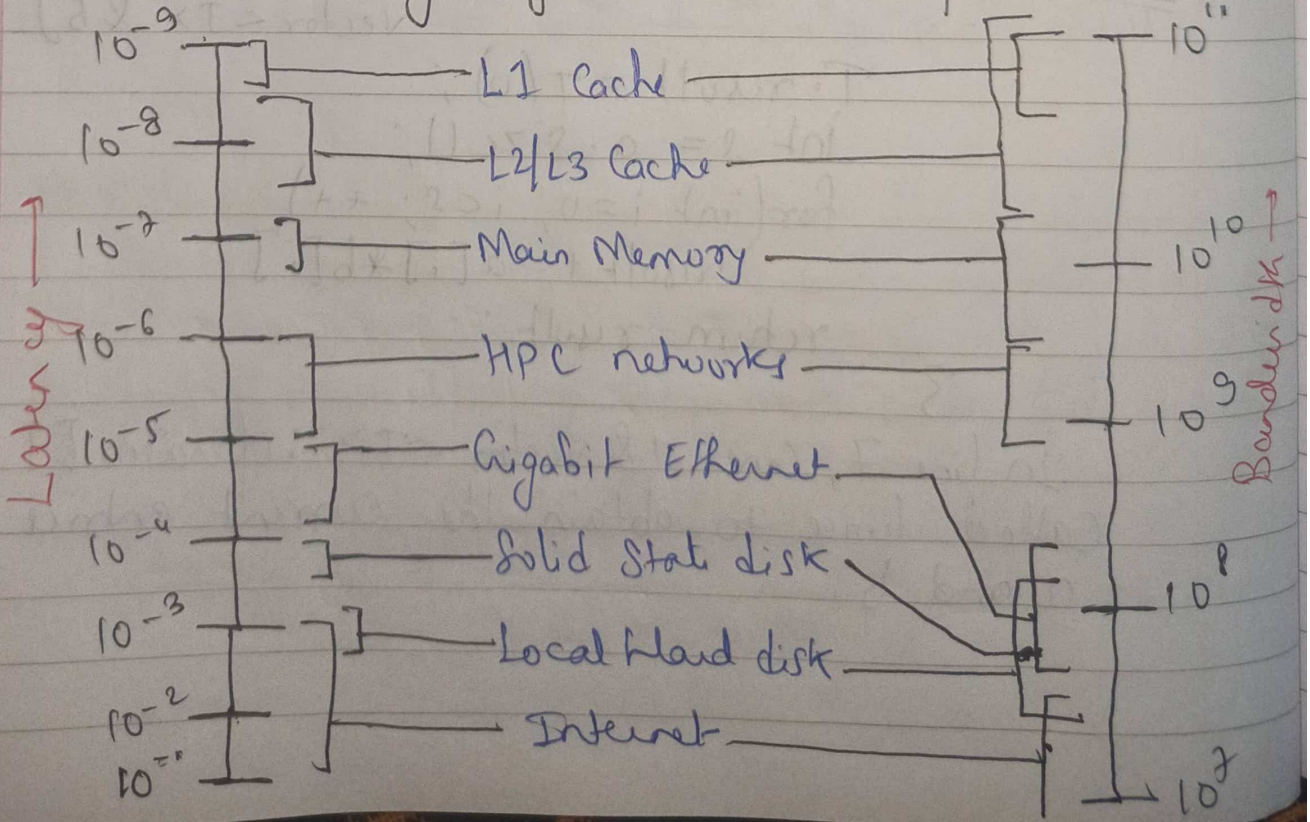
Of all porible performance-limiting factors in HPC, the most important one is data accen.

It's first aim is to reduce the traffic and making the data transfer as efficient as porible. (i.e Reduce traffic at slow paths).

DAO

- row-major
  - C program typically adopted row major Storage order.
- Column-major
  - FORTRAIN mainly use Column-major Storage order.

fig. of Data Accen optimization



We measure (i) Bandwidth (bytes/sec)
(ii) Latency (sec)
we get idea about performance of code.

Deeper the access of data, → increase in no. of levels we have to goes across.

Balance Analysis :-

Some programmers go to great length trying to improve the efficiency of code.
In order to decide whether this makes sense or if the program at hand is already using the resource in best possible way, one can often compute theoritical performance of loop-based code that is bounded by bandwidth limitations by simple rules of thumb.
The central concept to introduce here is balance.

→ Machine Balance (B_m) of a processor chip is the ratio of possible memory bandwidth in GWords/sec to peak performance in GFlops/sec.

$$B_m = \frac{\text{memory bandwidth (GWords/s)}}{\text{peak performance (GFlops/s)}} = \frac{b_{max}}{P_{max}}$$

data path	Balance
→ Cache	0.5 - 1.0
→ machine (memory)	0.03 - 0.5
→ interconnect (high speed)	0.001 - 0.02
→ interconnect (GBit ethernet)	0.0001 - 0.0007
→ disk (or disk subsystem	0.0001 - 0.01

Code Balance → In order to quantify the requirement of some code that runs on a machine with a certain balance, we further define code balance of a loop to be

$$B_c = \frac{\text{data traffic (words)}}{\text{floating point ops [Flops]}}$$

Reciprocal of Codebalance is often called computational intensity.

Using these 2 formula we can calculate the maximum performance of a loop.

$$P = \min\left(P_{max}, \frac{B_{max}}{B_c}\right)$$

P_{max} ← maximum performance

$$I = \min\left(1, \frac{B_{max}}{B_c}\right)$$

I ← Lightspeed of loop.

Ex →
```
for (i=0; i<n; i++) {
    A[i] = B[i] + C[i] * D[i];
}
```

$$\Rightarrow B_c = \frac{3+1}{2} = \frac{4}{2} = 2.$$

3 load (B, C, D), 1 store (A), 2 operation (+, *)

The STREAM Benchmarks

- The McCalpin STREAM benchmarks is a collection of four simple synthetic kernel loops which are supposed to fathom the capabilities of a processor's or a system's memory interface.
- Performance is usually reported as bandwidth in GBytes/sec.
- The STREAM TRIAD kernel is not to be confused with the vector triad, which has one additional load stream.
- The benchmarks exist in serial and OpenMP-parallel variants and are usually run with data sets large enough so that performance is safely memory bound.
- Measured bandwidth thus depends on the number of load and store streams only, and the results for COPY and SCALE (and likewise for ADD and TRIAD)

The STREAM Benchmarks

type	kernel	DP words	flops	B_c
COPY	A(:)=B(:)	2 (3)	0	N/A
SCALE	A(:)=s*B(:)	2 (3)	1	2.0 (3.0)
ADD	A(:)=B(:)+C(:)	3 (4)	1	3.0 (4.0)
TRIAD	A(:)=B(:)+s*C(:)	3 (4)	2	1.5 (2.0)

- This is important because optimizing compilers can recognize the STREAM source and substitute the kernels by hand-tuned machine code.
- Therefore, it is safe to state that STREAM performance results reflect the true capabilities of the hardware.
- They are published for many historical and contemporary systems on the STREAM Web site.
- Unfortunately, STREAM as well as the vector triad often fail to reach the performance levels predicted by balance analysis, in particular on commodity (PC-based) hardware.

The STREAM Benchmarks

The reasons for this failure are manifold and cannot be discussed here in full detail; typical factors are:

1. Maximum bandwidth is often not available in both directions (read and write) concurrently.

2. Protocol overhead, deficiencies in chipsets, error correcting memory chips, and large latencies (that cannot be hidden completely by pre-fetching) all cut on available bandwidth.

3. Data paths inside the processor chip, e.g., connections between L1 cache and registers, can be unidirectional.