CS528
High Performance Computing

Serial Code Optimization

A Sahu
Dept of CSE, IIT Guwahati

Outline

- Intro to Code Optimization
- Machine independent/dependent optimization
- Common sense of Optimization
  - Do less work, avoid expensive Ops, shrink working set
- Simple measure Large impact : simd, branch, comm sub expre
- C++ Optimization
- Scalar Profiling
    - Manual Instrumentation (get_wall_time, clock_t)
    - Function and line based profiling (gprof, gcov)
    - Memory Profiling (valgrind, callgraph)
    - Hardware Performance Counter (oprofile,likwid)

Profiling for Serial Code

Profiling for Serial Code

- Manual Instrumentation (get_wall_time, clock_t)
- Function and line based profiling (gprof, gcov)
- Memory Profiling (valgrind, callgraph)
- Hardware Performance Counter (oprofile,likwid)

Manual Instrumentation

- System Status
  - $uptime, $top , $vmstat
  - $systemmonitor, $gnome-system-monitor
- $time ./a.out
  - real time/wall clock time
  - cpu time and system time
  - cputime=sys time+usr time
- Using get_wall_time, clock_t

Manual Instrumentation

- $time command and Using get_wall_time,

```
#include <time.h>
int main(){
clock_t t; double Etime;
t = clock();
//Do some Work
t = clock() - t;
Etime=((double)t)/CLOCKS_PER_SEC;
printf("ETime =%f seconds",Etime)
return 0;
}
```

Profiler: Hotspot Analyzer

- Given a program
- Finding out part of the program which takes maximum amount of time
- Optimizing hot-spot area reduce the execution time significantly
- Suppose a program spend 99% of time in a small function/code
  - Optimizing that code will result better performance

Function and line based profiling

- GNU profile (gprof)
  - $gcc –p test.c
  - $./a.out
  - $gprof ./a.out
  - $gprof ./a.out >FPprofile.txt
- GNU coverage (gcov)

Gprof Example

```c
#include <stdio.h>
void FunA() {
   int i=0,g=0;
   while(i++<100000)
     {  g+=i; }
}
void FunB() {
   int i=0,g=0;
   while(i++<400000)
     { g+=i; }
}
```

```c
int main(){
    int iter=5000;
    while(iter--) {
        FunA();
        FunB();
    }
    return 0;
}
```

Gprof Example: Flat Profile

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
80.26	5.55	5.55	5000	1.11	1.11	FunB
20.94	6.99	1.45	5000	0.29	0.29	FunA

Gprof Example: Call Graph

```
Call graph
index % time    self  children    called     name
                                     <spontaneous>
[1]    100.0    0.00    6.99                  main [1]
                5.55    0.00    5000/5000       FunB [2]
                1.45    0.00    5000/5000       FunA [3]
-----------------------------------------------
                5.55    0.00    5000/5000       main [1]
[2]     79.3    5.55    0.00    5000        FunB [2]
-----------------------------------------------
                1.45    0.00    5000/5000       main [1]
[3]     20.7    1.45    0.00    5000        FunA [3]
```

Function and line based profiling

- GNU profile (gprof)
- GNU coverage (gcov)
  - **$gcc -fprofile-arcs -ftest-coverage tmp.c**
  - **$./a.out**
  - **$gcov tmp.c**

File 'tmp.c'

Lines executed:87.50% of 8

Creating 'tmp.c.gcov'

Gcov output

```
#include <stdio.h>
int main (){
   int i, total;
   total = 0;
   for (i = 0; i < 10; i++)
          total += i;
   if (total != 45)
          printf ("Failure\n");
   else printf ("Success\n");
   return 0;
}
```

```
        -:    1:#include <stdio.h>
        1:    2:int main (){
        -:    3:    int i, total;
        1:    4:    total = 0;
       11:    5:    for (i = 0; i < 10; i++)
       10:    6:        total += i;
        1:    7:    if (total != 45)
    #####:    8:        printf ("Failure\n");
        1:    9:    else printf ("Success\n");
        1:   10:    return 0;
        -:   11:}
```

Valgrind

- Free tools: **$sudo apt-get install valgrind**
- CallGraph, Profiler, Memory Check…
  - Many more
  - From C code, one can use API of valgrind
- Program analysis tools are useful
  - Bug detectors, Profilers, Visualizers
- **Dynamic binary analysis** (DBA) tools
  - Analyse a program's machine code at run-time
  - Augment original code with **analysis code**

Valgrind

```
void Work1(int n) {
    int i=0,j=0,k=0;
    while(i++<n){
        while(j++<n){while(k++<n);}
    }
}
void Work2(int n) { int i=0;  while(i++<n);}
void Maneger(int n1, int n2){
      Work1(n1); Work2(n2);
}
void Projects1() { Maneger(1000000, 1000);}
void Projects2() { Maneger(100, 1000000);}

int main() {
  Projects1(); Projects2();  return 0;
}
```

Valgrind: How to use

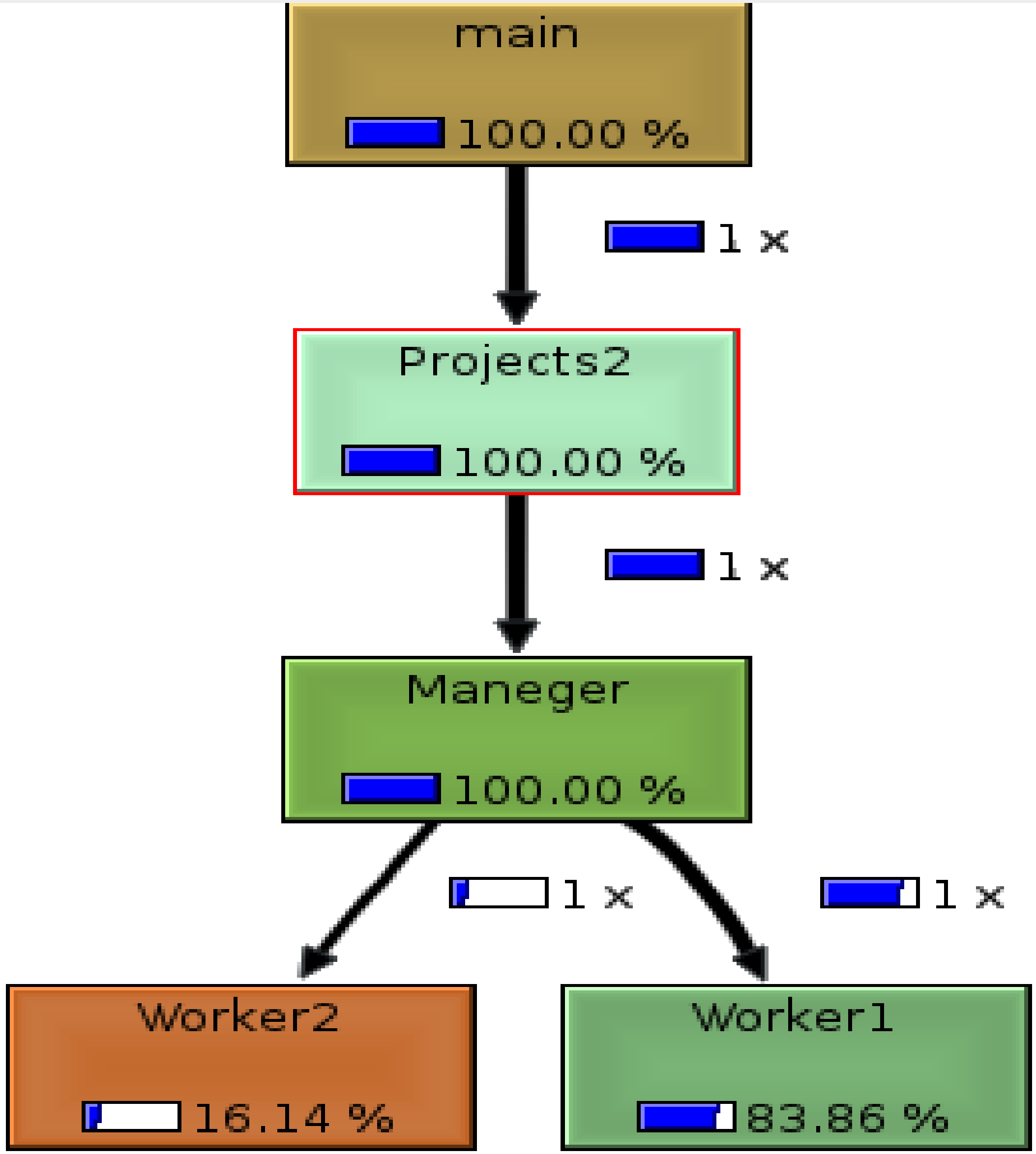
- $gcc -pg -o Valgrindtest Valgrindtest.c
- $valgrind --tool=callgrind ./Valgrindtest
- $ls

  Valgrindtest Valgrindtest.c callgrind.out.**11233**

$kcachegrind `ls -tr callgrind.out.* | tail -1`

pid

Valgrind: Call Graph



Further Optimizations for Serial Code

- Simple measure Large impact : simd, branch, comm sub expre
- C++ Optimization

Simple measures, large impact

- Elimination of Common Sub-expressions
- Avoid Branches:
  - **Code Can be SIMdized by compiler/gcc**
  - **Effective use of pipeline for loop code**
- Use of SIMD Instruction sets
  - **512 bit AVX SIMD in modern processor**
  - **ML/AI app use 8 bit Ops, can be speed up 512/8=64 time by simply SIMD-AVX**

Elimination of Common Sub-expressions

```
//value of s, r, x don't change in this loop
for (i=0; i<ALargeN; i++){
    A[i]=A[i]+s+r+sinx(x);
}
```

```
//value of s, r, x don't change in this loop
Tmp=s+r+sinx(x);
for (i=0; i<ALargeN; i++){
    A[i]=A[i]+Tmp;
}
```

Avoid Branches

```
for (i=0; i<N; i++)
    for(j=0;j<N;j++){
      if(i<j) S=1; else S=-1;
       C[i] =C[i]+S*A[i][j]*B[i];
    }
```

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

Use of SIMD: independent loop iteration

```
for (i=0; i<ALargeN; i++){
    A[i]=A[i]+B[i]*D[i];
}
All iterations in this loop are
independent : gcc SIMD utilize very
nicely
//ML application uses 8 bit OPS, 512
bit AVX SIMD 512/8=64 OPS can be done
in  parallel.
```

The ith iteration access : A[i], B[i], D[i]

Use of SIMD: independent loop iteration

$$S = \sum_{i=0}^{N} w_i x_i$$

- Vector dot product : is the most ***common and frequent kernel*** in
  - Matrix multiplication,
  - Neuron calculation (neural network NN)
    - Conv NN, Deep NN, //ML domain
  - Digital Signal Processing, Image Sig Processing, etc
  - Media Applications : audio, video, JPG/MPG, DCT.

Use of SIMD: independent loop iteration

$$S = \sum_{i=0}^{N} w_i x_i$$

```
for (i=0; i<ALargeN; i++){
    S=S+A[i]+B[i];
}
All iterations in this loop are
independent : gcc SIMD utilize very
nicely
//ML application uses 8 bit OPS, 512 bit
AVX SIMD 512/8=64 OPS can be done in
parallel.
```

The ith iteration access : A[i], B[i]

Use of SIMD: independent loop iteration

```
for (i=0; i<N; i++){
    A[i]=A[i]+B[i]; //S1
    B[i+1]=C[i]+D[i]; //S2
}
```

The ith iteration access : A[i], B[i], B[i+1],C[i] D[i]

Dependent loop iteration : i and i+1

Use of SIMD: independent loop iteration

```
A[0]=A[0]+B[0];
for (i=0; i<N; i++){
   B[i+1]=C[i]+D[i]; //S2
   A[i+1]=A[i+1]+B[i+1];//S1
}
B[N]=C[N-1]+D[N-1];
```

The ith iteration access :
A[i+1], B[i+1],C[i], D[i]

Use of SIMD: independent loop iteration

```
for (i=0; i<N; i++){
    A[i]=A[i]+B[i]; //S1
    B[i+1]=C[i]+D[i]; //S2
}
```

```
A[0]=A[0]+B[0];
for (i=0; i<N; i++){
    B[i+1]=C[i]+D[i]; //S2
    A[i+1]=A[i+1]+B[i+1];//S1
}
B[N]=C[N-1]+D[N-1];
```

Use of SIMD: independent loop iteration

- Affine access : index a.x+b form

```
for (i=0; i<N; i++){
    X[a*i+b]=X[c*i+d];
    //where a,b,c,d are integer
}
```

- GCD(c,a) divides (d-b) for loop dependence

- Ref Book : Hennesy Paterson, Advanced Computer Architecture, 5th Edition Book,

GCD test Example

```
for (i=0; i<N; i++){
        X[2*i+3]=X[2*i]+5.0;
      //X[a*i+b]=X[c*i+d];
}
```

- GCD(c,a) must divides (d-b) for loop dependence
- Value of a=2, b=3, c=2, d=0;
- GCD(a,c)=2, d-b=-3
- 2 does not divide -3 ➔ No dependence Possible