Day0

Table of Contents

1. The Syllabus

1.1. Uniprocessor optimization and Pthread

- Compiler level optimizations
- Arithmetic optimizations
- loop unrolling
- loop fusion
- loop interchange
- Loop invariant code motion
- Inlining
- Pthreads
- Overview
- What is thread
- What are Pthreads

- Why Pthreads
- Designing Threaded Programs
- Pthreads APIs
- Creating and Terminating Threads
- Passing Arguments to Threads
- Joining and Detaching Threads
- Stack Management
- Miscellaneous Routines
- Mutex Variables, and Condition Variables
- DPDK /SPDK libraries

1.2. MPI

- Basic MPI
- MPI Point to Point
- Type Matching
- MPI Collective Communication
    - Data Synchronization
    - Data Movement
    - Collective Computation
- Advanced MPI
    - Derived Data Types
    - Derived Types
- Special type Constructors
- Packing/Unpacking
- Data, Groups, and Communicators: Virtual Topologies
- MPI3 Standard, and MPI Threads
- Case Studies (Algorithms and Parallelization Approaches)
- Micro benchmarking, macro benching
- Application benchmarking and check the scalability of the applications

2. Compiler Optimization Demo

2.1. demo1.c

2.1.1. code

```c
#include<stdio.h>
int main(){
    int y = 3;
    int x = y + 6;
    printf("x = %d\n", x);
    return 0;
}
```

2.1.2. compile

```
file=demo1
gcc $file.c -O2 -o $file.out
#gcc -E $file.c > "$file_generated".c
```

```
file=demo1
#gcc $file.c -O2 -fdump-tree-all -o $file.out
gcc $file.c -O2 -fdump-tree-optimized
cat a-$file*.optimized
#gcc -E $file.c > "$file_generated".c
```

```
;; Function main (main, funcdef_no=23, decl_uid=3375, cgraph_uid=24, symbol_order=23) (ex

int main ()
{
  <bb 2> [local count: 1073741824]:
  __printf_chk (2, "x = %d\n", 9);
  return 0;

}
```

2.1.3. run

```
file=demo1
./$file.out
```

```
x = 9
```

2.2. demo2.c

2.2.1. code

```
#include<stdio.h>
inline int square(int x) {
    return x * x;
}
int main(){
    int x = 10;
    int y = square(x);
    printf("Square of %d : %d\n", x, y);
    return 0;
}
```

2.2.2. compile

```
file=demo2
gcc -O2 $file.c -o $file.out
```

```
file=demo2
#gcc $file.c -O2 -o $file.out
gcc $file.c -O2 -fdump-tree-optimized
cat a-$file*.optimized
```

```
;; Function main (main, funcdef_no=24, decl_uid=3378, cgraph_uid=25, symbol_order=24) (ex

int main ()
{
  <bb 2> [local count: 1073741824]:
  __printf_chk (2, "Square of %d : %d\n", 10, 100);
  return 0;

}
```

2.2.3. run

```
file=demo2
./$file.out
```

```
Square of 10 : 100
```

3. Arithmetic Optimizations

3.1. Constant folding

Simplifies constant expressions at compile time, reducing runtime calculations.

```
int x = 3 + 10;
```

```
int x = 13;
```

3.2. Constant Propagation

Replaces variables with constant values if they are known at compile time, enabling further optimizations.

```
int x = 10;
int y = x + 3;
```

```
int x = 10;
int y = 10 + 3;
// which eventually will become
// int y = 13;
```

3.3. Strength Reduction

Replaces expensive operations with cheaper ones.

3.3.1. Multiplication to Bitwise

```
x * 8;
```

```
x << 3;
```

3.3.2. Division to Multiplication

```
x / 2;
```

```
x * 0.5;
```

3.4. Algebraic Simplifications

Simplifies algebraic expressions to more efficient forms.

3.4.1. Removing Common Subexpressions

```
a * (b + c) + d * (b + c);
```

```
(a + d) * (b + c);
```

3.4.2. Simplifying Arithmetic

```
x + 0;
y * 1;
```

```
x;
y;
```

4. Loop Optimizations

4.1. Loop Unrolling

Increases the loop body size by replicating it multiple times, reducing the overhead of loop control.

4.1.1. Example 1

```
for (int i = 0; i < 4; i++) {
    // Loop body
}
```

```
//Loop body
//Loop body
```

```
//Loop body
//Loop body
```

4.1.2. Example 2

```
int arr[N];
for (int i = 0; i < N; i++) {
    sum += arr[i];
}
```

- N iterations required

```
int arr[N];
for (int i = 0; i < N - 1; i+=2) {
    sum += arr[i];
    sum += arr[i + 1];
}
```

- N/2 iterations required

4.2. Loop Fusion

Merges adjacent loops with the same iteration range into a single loop.

```
int x = 0;
int y = 0;
for (int i = 0; i < n; i++) {
    x++;
}
for (int i = 0; i < n; i++) {
    y++;
}
printf("x = %d\n", x);
printf("y = %d\n", y);
```

```
int x = 0;
int y = 0;
for (int i = 0; i < n; i++) {
    x++;
    y++;
}
printf("x = %d\n", x);
printf("y = %d\n", y);
```

4.3. Loop Interchange

Swaps inner and outer loops to improve cache performance.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        // Loop body
    }
}
```

```
for (int j = 0; j < m; j++) {
    for (int i = 0; i < n; i++) {
        // Loop body
    }
}
```

4.4. Loop Invariant Code Motion

Moves code that does not change within the loop outside of the loop.

```
int y = 0;
for (int i = 0; i < n; i++) {
    int x = 5; // Invariant code
    y+= x;
}
printf("y = %d\n", y);
```

```
int x = 5;
for (int i = 0; i < n; i++) {
    y+= x;
}
printf("y = %d\n", y);
```

5. Function Inlining

Replaces a function call with the function's code to avoid the overhead of a call and return.

```
inline int square(int x) {
    return x * x;
}

int y = square(5);
```

```
int y = 5 * 5;
```

6. Dead Code Elimination

Removes code that will never be executed or whose results are never used.

```
int main(){
    if (false) {
    // Dead code
    }
    int x = 10;
    printf("x = %d\n",x);
    return x;
    x = 20; // Dead code
}
```

```
int main(){
```

```
    int x = 10;
    printf("x = %d\n",x);
    return x;
}
```

Author: Abhishek Raj
Created: 2024-06-25 Tue 15:11