Cost Metric for Evaluation of Parallel OpenMP Codes

CSE 530 Project Presentation

Bodhisatwa Chatterjee
bxc583@psu.edu

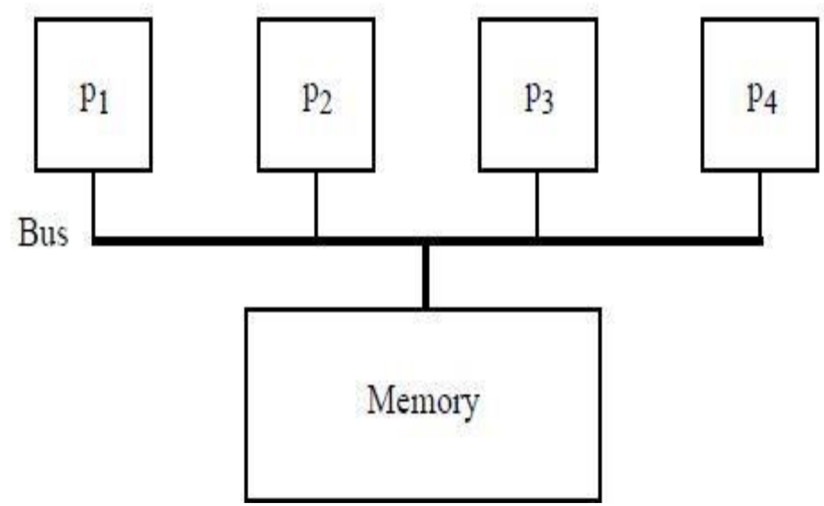
Outline of this presentation

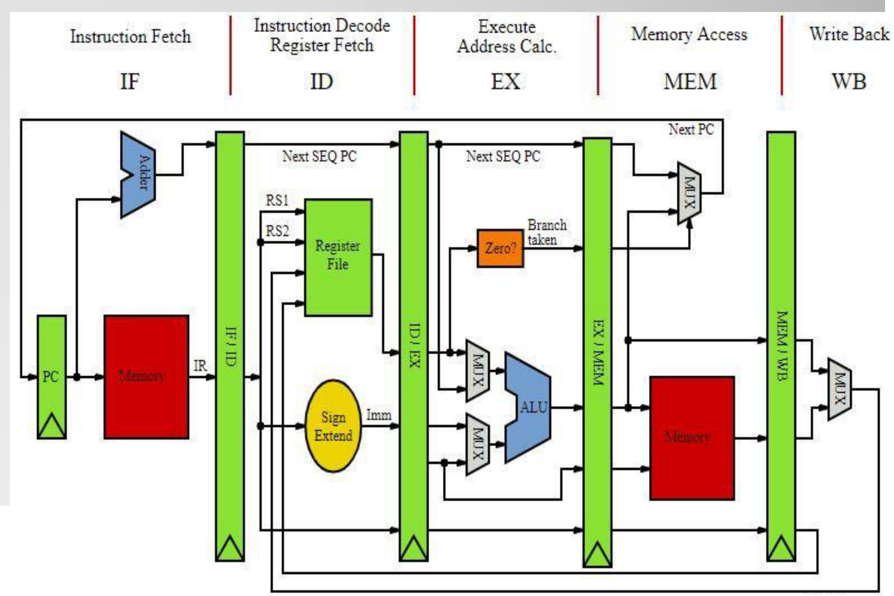
- Parallelism Overview
- Problem of getting multiple parallel versions of a sequential problem
- Cost Metric to distinguish these versions

Parallelism is Indispensable

- Pipelining was the earliest application
- Processor Parallelism (Synchronous & Asynchronous)
- Multiple functional units, multiple issue instruction units to facilitate parallel execution

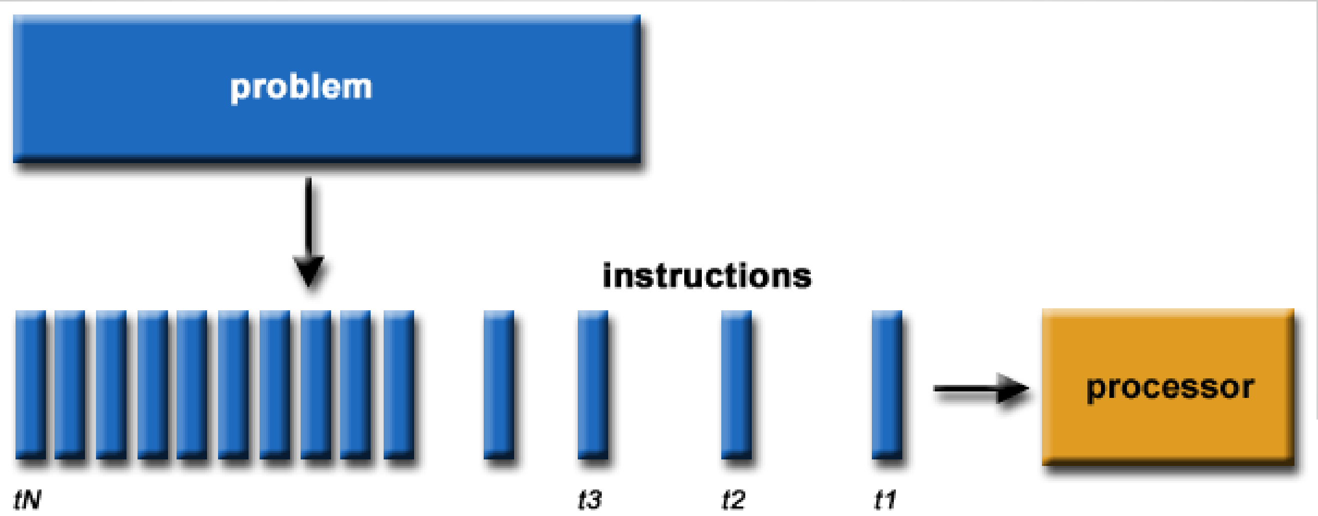
Asynchronous shared-memory multiprocessor.



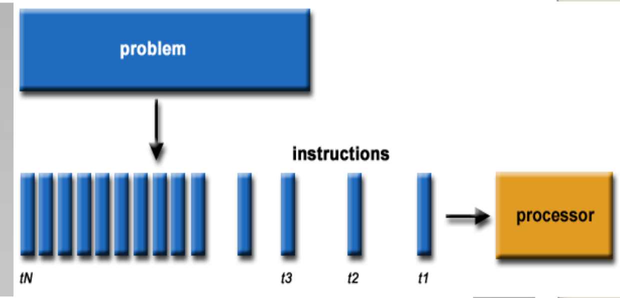


Need to utilize these resources

- Having fast hardware is not sufficient
- Need to have software taking advantage of these resources
- Need to make sure that our code exhibits parallel behaviour



Scope for Parallelism in Workloads?




```
void FLOPSBenchmark()
{
        long l, c, m, id;
        double fs_t, fe_t, ft_t;
        struct fth ft[th_count];

        for(l =0; l < th_count; l++)
        {
                ft[l].lc = 1;
                ft[l].th_counter = 1;
                ft[l].fa = 0.02;
                ft[l].fb = 0.2;
                ft[l].fc = 0;
                ft[l].fd = 0;
        }

        gettimeofday(&t, NULL);
        fs_t = t.tv_sec+(t.tv_usec/1000000.0)

        for(c = 0; c < th_count; c++)
        {
                pthread_create(&ft[c].threads
        }
        for(m =0; m < th_count; m++)
        {
                pthread_join(ft[m].threads, N
        }

        gettimeofday(&t, NULL);
        fe_t = t.tv_sec+(t.tv_usec/1000000.0)
        ft_t = fe_t - fs_t;
        f_avg += (loop_count*30) / (ft_t * 10
```

```
// FlOPs Be
void *FAdd(
{
        stru
        th_d

        for(
```

```
// GPU FLOPS
__global__ v
{
        int i

        if (i
        {
```

```
 * of the todo queue.
 *
 * Requires the proc->inner_lock to be held.
 */
static void
binder_enqueue_thread_work_ilocked(struct binder_thread *thread,
                                   struct binder_work *work)
{
        WARN_ON(!list_empty(&thread->waiting_thread_node));
        binder_enqueue_work_ilocked(work, &thread->todo);
        thread->process_todo = true;
```

FUNCTIONS?

DATA STRUCTURE ACCESSES?

```
 * binder_enqueue_thread_work() - Add an item to the thread work list
 read:        thread to queue work to
 rk:          struct binder_work to add to list

 e work to the todo list of the thread, and enables processing
 todo queue.

 eue_thread_work(struct binder_thread *thread,
                 struct binder_work *work)

 ler_inner_proc_lock(thread->proc);
 ler_enqueue_thread_work_ilocked(thread, work);
 ler_inner_proc_unlock(thread->proc);
```

LOOPS?!

Focus is to parallelize <u>LOOPS</u>

- Owing to their iterative nature

<u>Toy Example</u>

$$for(i = 0; i \leq 10; i++)$$
$$A[i] = i$$

- Basic idea is to execute each iteration in parallel
- <u>Parallel Execution:</u>

i = 0	i = 1	....	i = 10
A[0]=0	A[1]=1		A[10]=10
↓	↓		↓
P0	P1		P10

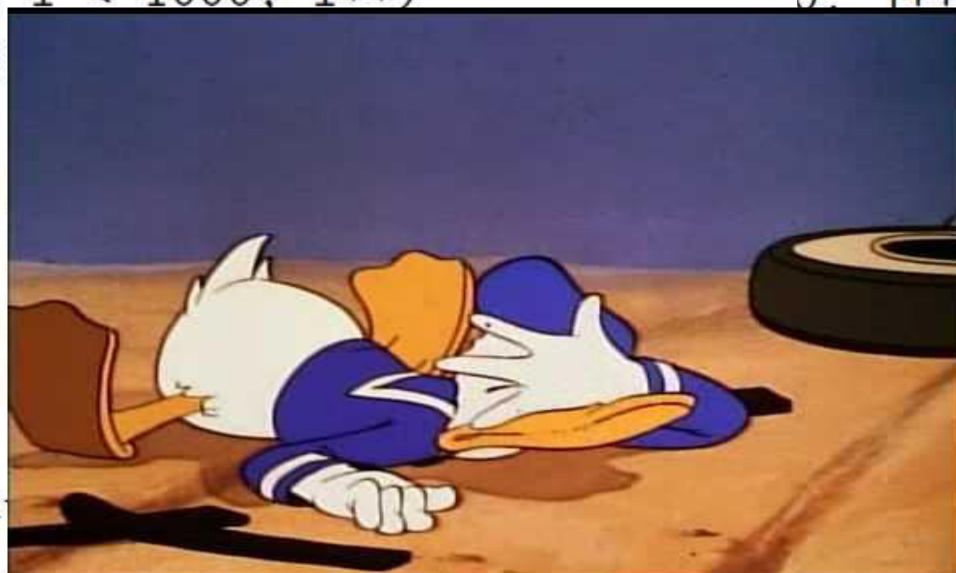
Is life always that easy?

- Dependencies!!

```
for ( i = 0; i < 1000; i++)
  for (j = 0
  {
      a[i+1]
      b[i+2]
      c[i+2]
      d[i+2]
  }
```

```
for ( i = 0; i < 1000; i++)
                    0; j++)

                    +2][j];
                    2*j;
                    + e[i][j];
                    + 2*j;
```



```
                    ][j-1];
```

```
for(i=0; i                i++)
{        A[i
         B[i] = 2+i;          [i] = A[i+5]+ 2*i;
}                             [i] = B[i]+A[i] + 4;
                          }
```

Need transformations to resolve dependencies whenever possible

- Loop interchange, Loop normalization, Loop unrolling, etc

```
      DO I = 1, N
         DO J = 1, M
S           A(I,J+1) = A(I,J) + B
         ENDDO
      ENDDO
```

```
      DO J = 1, M
         DO I = 1, N
S           A(I,J+1) = A(I,J) + B
         ENDDO
      ENDDO
```

- Simple transformation enabled parallelization of the inner loop.

End up with multiple versions of parallel codes

- Parallel codes depends on specific loop transformations and the order in which they are applied
- Types of transformations and their order depends upon the parallel framework/model (Polyhedral Model, Unimodular framework, Presbuyer Framework) and different tools use different frameworks
- Frameworks are high level abstractions to perform loop transformations

Distinguishing multiple versions of parallel codes

```
/* Start of CLooG code */
#omp pragma parallel
C[3] = A[3 -3] + 20;;
for (t1=4;t1<=7;t1++) {
  C[t1] = A[t1-3] + 20;;
  B[(t1-1)] = A[(t1-1)]
  D[(t1-1)] = B[(t1-1)] + C[(t1-1)];;
}
for (t1=8;t1<=1999;t1++) {
  C[t1] = A[t1-3] + 20;;
  A[(t1-5)+4] = 3*(t1-5)
  B[(t1-1)] = A[(t1-1)] +
  D[(t1-1)] = B[(t1-1)] +
}
A[1995 +4] = 3*1995 + j + 1;;
B[1999] = A[1999] + 10;;
D[1999] = B[1999] + C[1999];;
for (t1=2001;t1<=2004;t1++)
  A[(t1-5)+4] = 3*(t1-5) + j
}
```

Parallel Loops

Sequential Loops

Vector Loops

```
/* Start of CLooG code */
#omp pragma parallel
for (t1=3;t1<=4;t1++) {
        = A[t1-3] + 20;;
        = D[t1] + 10;;
  D[t1] = B[t1-3] + C[t1];;
}
for (t1=5;t1<=1999;t1++) {
  C[t1] = A[t1-3] + 20;;
          D[t1] + 10;;
          B[t1-3] + C[t1];;
        +4] = 3*(t1-2) + C[(t1-2)-2];;
}
#pragma ivdep
#pragma vector always
for (t1=2000;t1<=2001;t1++) {
     2)+4] = 3*(t1-2) + C[(t1-2)-2];;
```

Cost Metric

- Basic Idea: Need to find the number of iterations of each type of loops statically
- Each iteration is defined as an unit cost
- Cost Metric has number of iterations of each loops and a synchronization cost as its parameters

```
/* Start of CLooG code */
lbp=0;
ubp=99;
#pragma omp parallel for private(lbv,ubv,t2)
for (t1=lbp;t1<=ubp;t1++) {
  lbv=0;
  ubv=10;
#pragma ivdep
#pragma vector always
  for (t2=lbv;t2<ubv;t2++) {
```

PLuTo. The parallel loop will run from 0 to 99 and assuming a quadcore machine, we get $Cost_{Parallel} = (100/4) = 25$. Similarly,$Cost_{Vector} = (10/4) = 2.5$. Therefore, cost metric output tuple of this code is : (0,25,2.5,1)

Calculating Iterations is not always easy

- Loops with functional bounds
- Often bounds are not known at compile time
- Derived two novel methods to calculate iterations of loops with functional bounds

$$\textbf{for}\ (i = 0;\ i < 10;\ i{+}{+})$$
$$\quad \textbf{for}\ (j = max(2*i+4, 10);\ j <= min(3*i, 2);\ j{+}{+})$$
$$\qquad S_1$$

Figure 1.3 Example of a sequential loop having functional bounds generated by PLuTo - The functions used here are **max** & **min**, which are pre-defined as Macros in the generated codes.

Point of Inflection and Critical Point Analysis

- Two methods which deal with functional bounds
- Point of Inflection is used to double level nested loop with functional bounds
- Critical point analysis deals with n-level nesting with algebraic bounds

for $(I_1 = L; I_1 <= U; I_1++)$
  for $(I_2 = f(a*I_1+b, k_1); I_2 <= f'(c*I_1+d, k_2);$
  $I_2++)$
    $S_1;$

Figure 1.3 Example of a General *Category I* loop - functions f and f' are defined as Macros in the generated parallel OpenMP Codes and are generally restricted to max, min, floor and ceil.

for $(I_1 = L_1; I_1 \leq U_1; I_1++)$
  for $(I_2 = f(I_1); I_2 \leq f'(I_1); I_2++)$
  .
  .
  .
  .
  for $(I_k = f(I_1, I_2, \ldots, I_{k-1}); I_k \leq f'(I_1, I_2, \ldots, I_{k-1});$
  $I_k++)$
    S_1

Figure 1.2 Generic form of a simple Category 2 sequential loop - This loop contains algebraic bounds which are obtained by transformation from the original functional bounds.

Future Work

- Addition of more parameters to the cost metric (function call, arrays as pointers)
- Design of a tool to calculate cost metrics of OpenMP Codes