Computational complexity of multithreaded algorithms

Prof Dr Marko Robnik-Šikonja

Analysis of Algorithms and Heuristic Problem Solving
November 2019
Architecture of multi-core processor
Shared and distributed memory

(a) Shared Memory
- Cache
- Cache
- Cache
- Cache-Coherent Interconnect

(b) Non-Cache-Coherent Interconnect
- P
- P
- P
- Memory

(c) Distributed Shared Memory
- P
- P
- P
- Memory
- Memory
- Memory

(d) Network
- P
- P
- P
- Memory
- Memory
- Memory
- ...
Problems with parallel execution

* deadlock: each member of a group is waiting for another member, including itself, to take action

* happens under the following Coffman conditions
  * mutual exclusion: at least one resource must be held in a non-shareable mode
  * hold and wait (resource holding): a process is holding at least one resource and requesting additional resources which are being held by other processes.
  * no preemption: a resource can be released only voluntarily by the process holding it.
  * circular wait: each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource

* livelock: two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work.

* starvation: some resource may always be allocated to some process

* race conditions and synchronization: system attempts to perform two or more operations at the same time, but the operations must be done in the proper sequence to be done correctly.
Race conditions

* deterministic and nondeterministic multithreaded programs

```c
void Race() {
    int x = 0;
    parallel for i=1 to 2
        x = x +1;
    print x;
}
```
Low level synchronization mechanisms

* monitor: a mechanism that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become false. Monitors also have a mechanism for signaling other threads that their condition has been met.

* semaphore: a (counting) variable controlling access to a common resource

* atomic operations: program operations that cannot be preempted
Dynamic threads

* simplified programming,
* top-level parallelism
* three new constructs parallel, spawn, sync
* simplified complexity analysis
* platforms: Cilk, Cilk++, OpenMP, Task Parallel Library (.NET), Threading Building Blocks(C++, Intel), JOMP, JPPF (Java)
OMP elements

OpenMP language extensions

- parallel control structures
  - parallel directive
  - governs flow of control in the program
- work sharing
  - do/parallel do and section directives
  - distributes work among threads
- data environment
  - shared and private clauses
- synchronization
  - critical and atomic directives
  - barrier directive
  - coordinates thread execution
- runtime functions, env. variables
  - omp_set_num_threads()
  - omp_get_thread_num()
  - OMP_NUM_THREADS
  - OMP_SCHEDULE
  - runtime environment
Example: Fibonacci numbers

\[F_0 = 0\]
\[F_1 = 1\]
\[F_n = F_{n-1} + F_{n-2} \text{ za } n \geq 2\]

```c
int fib (int n){
    if (n <= 1)
        return n ;
    else {
        int x = fib(n-1) ;
        int y = fib(n-2) ;
        return x + y ;
    }
}
```

\[T(n) = T(n-1) + T(n-2) + \Theta(1)\]

\[\text{solution } T(n) = \Theta(\tau^n)\]
\[\tau = (1 + \sqrt{5})/2\]
Multithreaded Fibonacci

```c
int pFib (int n){
    if (n <= 1)
        return n ;
    else {
        int x = spawn pFib(n-1) ;
        int y = pFib(n-2) ;
        sync
        return x + y ;
    }
}
```

* nested parallelism
* scheduler
int pFib (int n){
    if (n <= 1)
        return n ;
    else {
        int x, y ;
        # pragma omp sections public(x, y)
        {
            #pragma omp section
            x = pFib(n-1) ;

            #pragma omp section
            y = pFib(n-2) ;
        }
        return x + y ;
    }
}
Multithreaded computational model

* acyclic directed graph
* equal processors
* no resources for scheduling
* total time, time of parallel tasks
* critical path
* number of processors $P$, time for $P$ processors $T_P$
* $T_1, T_\infty$
Parallel speedup

* in one step, using P processors, we finish P units of work, in time $T_P$ we do $P \cdot T_P$ units
* total work is $T_1$, note that $P \cdot T_P \geq T_1$
* speedup rule: $T_P \geq T_1 / P$
* also $T_P \geq T_\infty$
* speedup or level of parallelism is $T_1 / T_P \leq P$
* linear speedup $T_1 / T_P = \Theta(P)$
* ideal linear speedup $T_1 / T_P = P$
Analysis of parallel algorithms

Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$
Analysis of parallel algorithms

* Fibonacci
  * $T_1(n) = T_1(n-1) + T_1(n-2) + \Theta(1)$

* $T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1)$
  = $T_\infty(n-1) + \Theta(1)$
  = $\Theta(n)$
Limits of parallelization

* Amdahl’s law

- speedup $S = \frac{T_1}{T_P}$
- $f =$ proportion of parallelizable code

* $S = \frac{1}{\frac{f}{P} + (1-f)}$

* let us compute speedup for 2, 5, 10, $\infty$, processors and $f=0.9, 0.5, 0.1$
Amdahl’s law
Gustafson’s law

* Processing time (on each processor) is split to
  \( T_1 = a + b \) (a is sequential time, b = parallel time)
  Sequential share of work \( \alpha = \frac{a}{a+b} \)
  1 - \( \alpha \) is a share of parallel work

* Assumption: using more parallel units we can solve larger
  problems (or more problems in the same time), the size of
  problems grows linearly with \( P \), therefore \( T_1 = a + P\cdot b \)

* Speedup \( S_P = \frac{a + P\cdot b}{a+b} = \alpha + P(1 - \alpha) = P - \alpha (P - 1) \)

* For small \( \alpha \) the speedup is almost linear in \( P \)
Speedup by Gustafson

Gustafson’s Law: $S(P) = P \cdot \alpha \cdot (P-1)$

- $x - 0.1 \cdot (x-1)$
- $x - 0.2 \cdot (x-1)$
- $x - 0.3 \cdot (x-1)$
- $x - 0.4 \cdot (x-1)$
- $x - 0.5 \cdot (x-1)$
- $x - 0.6 \cdot (x-1)$
- $x - 0.7 \cdot (x-1)$
- $x - 0.8 \cdot (x-1)$
- $x - 0.9 \cdot (x-1)$
Two views of parallel speedup, Amdahl and Gustafson

* Amdahl: if we travel to destination 100km away and we used 1 hour for one half of the distance, the total average time will never reach 100km/h, no matter how fast we travel the second half.

* Gustafson: suppose you travel for some time with a speed lower than 100km/h; if the distance is long enough and there is enough time available you can still reach arbitrary average speed; e.g., if you travel 1 hour with the speed 50km/h and continue the next hour with 150km/h, the total average speed will be 100km/h (or you can travel next half an hour with 200km/h).
* Example: multiplication of a matrix and vector $y = A \times x$

```c
void mat_vec(matrix A, vector x) {
    int n = A.rows;
    // let the length of y be n
    for i = 1 to n
        y[i] = 0;
    for i = 1 to n
        for j = 1 to n
            y[i] = y[i] + A[i][j] * x[j];
    return y;
}
```

```c
void mat_vec(matrix A, vector x) {
    int n = A.rows;
    // let the length of y be n
    parallel for i = 1 to n
        y[i] = 0;
    parallel for i = 1 to n
        for j = 1 to n
            y[i] = y[i] + A[i][j] * x[j];
    return y;
}
```

high-level parallelization
Loop parallelization: actual schedule

// what code would compiler generate for main loop
void mat_vec_main_loop(matrix A, vector x, vector y, n, i, k) {
  if (i == k) {
    for j = 1 to n
      y_i = y_i + A_{ij} * x_j ;
  }
  else {
    mid = (i + k) / 2 ; // the floor
    spawn mat_vec_main_loop(A, x, y, n, i, mid)
    mat_vec_main_loop(A, x, y, n, mid+1, k)
    sync
  }
}

Compiler might generate more coarse parallelization
void wrong_mat_vec(matrix A, vector x) {
    int n = A.rows;
    parallel for i = 1 to n
        y_i = 0;
    parallel for i = 1 to n
        parallel for j = 1 to n
            y_i = y_i + A_{ij} * x_j;  // race !!
    return y;
}

Races