第四部分 并行程序设计

- 第十二章 并行程序设计基础
- 第十三章 共享存储并行编程
- 第十四章 分布存储并行编程
- 第十五章 并行程序开发环境

串行程序设计与并行程序设计

- 串行程序设计
- 并行程序设计困难的原因
  - 并行算法没有很好的范例
  - 计算模型不统一
  - 并行编程语言还不成熟完善
  - 环境和工具缺乏较长的生长期, 缺乏代可扩展和异构可扩展
- 并行程序设计的进展
  - 已有很多并行算法，且其中有一些好的范例
  - 编程类型：共享变量、消息传递、数据并行

并行程序设计的途径

- (a) 使用库例程构造并行程序
  - 编译代码段
  - for (i=0; i<N; i++) A[i]=b[i]*b[i+1];
  - for (i=0; i<N; i++) c[i]=A[i]+A[i+1];

- (b) 扩展串行语言
  - my_process_id, number_of_processes()
  - for (i=0; i<N; i++) A[i]=b[i]*b[i+1];
  - barrier();
  - for (i=0; i<N; i++) c[i]=A[i]+A[i+1];

- (c) 加编译器构造并行程序的方法
  - #pragma parallel
  - #pragma shared(A,b,c)
  - #pragma local(id)
  - { #pragma pfor iterate (i=0; N; 1)
    - for (i=0; i<N; i++) A[i]=b[i]*b[i+1];
    - #pragma synchronize
    - for (i=0; i<N; i++) c[i]=A[i]+A[i+1];
  - }

- (d) 自动并行化：Autopar

- (e) 设计新并行语言：Linda, Ocean
What is a parallel programming model?

- is the task assignment to the parallel computing unit (i.e., task), which is decided by the hardware model.
- The model decides how to effectively execute parallel tasks on the hardware.

Main Goal:

- Utilize all processors of the machine (e.g., SMP, MPP, NUMA) to minimize program execution time.

Shared Memory Model

- In the shared memory programming model, the abstraction is that parallel tasks can access any location of the memory.
- Parallel tasks can communicate through reading and writing common memory locations.
- This is similar to threads from a single process which share a single address space.
- Multi-threaded programs (e.g., OpenMP programs) are the best fit with the shared memory programming model.

**Parallel Programming Models**

- Shared Memory
- Message Passing

**Shared Memory Model**

- `S = Serial P = Parallel`
Shared Memory Example

```plaintext
begin parallel // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter=4, sum=0;
shared double sum=0.0, a[], b[], c[];
for (i=0; i<8; i++)
    if (a[i] > 0)
        sum = sum + a[i];
Print sum;
```

```plaintext
Shared Memory Example
```

```plaintext
begin parallel // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter=4, sum=0;
shared double sum=0.0, a[], b[], c[];
for (i=0; i<8; i++)
    if (a[i] > 0)
        sum = sum + a[i];
Print sum;
end parallel // kill the child thread
```

Message Passing Model

- In message passing, parallel tasks have their own local memories
- One task cannot access another task's memory
- Hence, to communicate data they have to rely on explicit messages sent to each other
- This is similar to the abstraction of processes which do not share an address space
- MPI programs are the best fit with message passing programming model

Message Passing Model

```
Parallel Programming Models

- Shared Memory
- Message Passing
```
Message Passing Example

```plaintext
for (i=0; i<8; i++)
id = getpid();
local_iter = 4;
start_iter = id * local_iter;
end_iter = start_iter + local_iter;
if (id == 0) send_msg(P1, b[4..7], c[4..7]);
else recv_msg(P0, b[4..7], c[4..7]);
for (i=start_iter; i<end_iter; i++)
a[i] = b[i] + c[i];
local_sum = 0;
for (i=start_iter; i<end_iter; i++)
if (a[i] > 0)
    local_sum = local_sum + a[i];
sum = local_sum;
if (id == 0) send_msg(P0, local_sum);
```

Shared Memory Vs. Message Passing

- **Shared Memory**
  - Implicit (via load/store)
  - Explicit (via Messages)

- **Message Passing**
  - Explicit (via Messages)
  - Implicit (via load/store)

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>Implicit</td>
<td>Explicit Messages</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Explicit</td>
<td>Implicit (Via Messages)</td>
</tr>
<tr>
<td>Hardware Support</td>
<td>Typically Required</td>
<td>None</td>
</tr>
<tr>
<td>Development Effort</td>
<td>Lower</td>
<td>Higher</td>
</tr>
<tr>
<td>Tuning Effort</td>
<td>Higher</td>
<td>Lower</td>
</tr>
</tbody>
</table>

MPMD-Programming Paradigm

- The MPMD model uses different programs for different processes, but the processes collaborate to solve the same problem.
- MPMD has two styles, the master/worker and the coupled analysis.

Example

```
a.out= Structural Analysis, b.out = fluid analysis and c.out = thermal analysis
```
例：Cannon矩阵乘法

- 适用于网格
- 存储有效
- 规则通信
  - 初始化
  - 对齐
  - 循环移位

17. Cannon乘法：初始化

18. Cannon乘法：对齐A

19. Cannon乘法：对齐B
每个结点并行乘分块得到C分块

各自乘分块并累加到C分块

A和B各自循环移位

Cannon矩阵乘法MPI程序

- 参数a, b, c分别指向局部存储的分块，分别对应A, B, C，均是n阶方阵。
- 处理器个数为p时分块为sqrt(n)阶方阵，p=2^t，t为正整数。
- 参数comm表示调用MatrixMatrixMultiply()的进程标识，通信簇。
- 函数接口 MatrixMatrixMultiply(int n, double *a, double *b, double *c, MPI_Comm comm)
MatrixMatrixMultiply(int n, double *a, double *b, double *c, MPI_Comm comm) {
    int i, nlocal, npes, dims[2], periods[2];
    int myrank, my2drank, mycoords[2];
    int uprank, downrank, leftrank, rightrank, coords[2];
    int shiftsrc, shiftdest;
    MPI_Status status;
    MPI_Comm comm_2d;

    /* Get the communicator related information */
    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);

    /* Set up the Cartesian topology */
    dims[0] = dims[1] = sqrt(npes);
    periods[0] = periods[1] = 1;
    MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);

    /* Get the rank and coordinates */
    MPI_Comm_rank(comm_2d, &my2drank);
    MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

    /* Determine the dimension of the local matrix block */
    nlocal = n/dims[0];

    /* Perform the initial matrix alignment. */
    MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsrc, &shiftdest);
    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE, shiftdest, 1, shiftsrc, 1, comm_2d, &status);
    MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsrc, &shiftdest);
    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE, shiftdest, 1, shiftsrc, 1, comm_2d, &status);

    /* Get into the main computation loop */
    for (i=0; i<dims[0]; i++) {
        MatrixMultiply(nlocal, a, b, c); /* c = c + a*b */
        MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
        MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
        MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE, uprank, 1, downrank, 1, comm_2d, &status);
        MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE, downrank, 1, uprank, 1, comm_2d, &status);
    }

    /* Restore the original distribution of a and b */
    MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsrc, &shiftdest);
    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE, shiftdest, 1, shiftsrc, 1, comm_2d, &status);
    MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsrc, &shiftdest);
    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE, shiftdest, 1, shiftsrc, 1, comm_2d, &status);

    MPI_Comm_free(&comm_2d); /* Free up communicator */
}

MatrixMultiply(int n, double *a, double *b, double *c) {
    int i, j, k;

    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            for (k=0; k<n; k++)
                c[i*n+j] += a[i*n+k]*b[k*n+j];
}

/* This function performs a serial matrix-matrix multiplication c = a*b */
MatrixMultiply(int n, double *a, double *b, double *c) {
    int i, j, k;

    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            for (k=0; k<n; k++)
                c[i*n+j] += a[i*n+k]*b[k*n+j];

并行编程范型：OpenMP Master-Worker

- 适用于MIMD-SM结构，如多核、GPU
- 线程级并行程序
- 主要机制：循环并行化、线程派生和交互机制

相并行（Phase Parallel）

- 一组超级步（相）
- 步内各自计算
- 步间通信、同步
- BSP（4.2.3）
- 方便查错和性能分析
- 计算和通信不能重叠

并行编程范型

- 相并行（Phase Parallel）
- 分治并行（Divide and Conquer Parallel）
- 流水线并行（Pipeline Parallel）
- 主从并行（Master-Slave Parallel）
- 工作池并行（Work Pool Parallel）
- Task-Farming
- SPMD
- Data Pipeline
- Divide and Conquer
- Speculative Parallelism

主－从并行（Master-Slave Parallel）

- Task-Farming
- 主进程：串行、协调任务
- 子进程：计算子任务
- 划分设计技术（6.1）
- 与相并行结合
- 主进程易成为瓶颈
分治并行（Divide and Conquer Parallel）
- 父进程把负载分割并指派给子进程
- 递归
- 重点在于归并
- 分治设计技术（6.2）
- 难以负载平衡

工作池并行（Work Pool Parallel）
- 初始状态：一件工作
- 进程从池中取任务执行
- 可产生新任务放回池中
- 至任务池为空
- 易于负载平衡
- 临界区问题（尤其消息传递）

流水线并行（Pipeline Parallel）
- 一组进程
- 流水线作业
- 流水线设计技术（6.5）

并行程序设计模型（续）
- 隐式并行（Implicit Parallel）
- 数据并行（Data Parallel）
- 共享变量（Shared Variable）
- 消息传递（Message Passing）
隐式并行（Implicit Parallel）

- 概况:
  - 程序员用熟悉的串行语言编程
  - 编译器或运行支持系统自动转化为并行代码

- 特点:
  - 语义简单
  - 可移植性好
  - 单线程，易于调试和验证正确性
  - 效率很低

数据并行（Data Parallel）

- 概况:
  - SIMD的自然模型
  - 局部计算和数据选路操作

- 特点:
  - 单线程
  - 并行操作于聚合数据结构（数组）
  - 松散同步
  - 单一地址空间
  - 隐式交互作用
  - 显式数据分布

Programming Model 2: Data Parallel

- 单线程控制（Single thread of control）
  - 包含并行操作
  - 并行操作应用于所有数据结构（或一个定义的子集）
  - 通常对数组

- 特点:
  - 语义简单
  - 易于理解和推理
  - 协调是隐含的

- 劣势:
  - 不适合所有问题
  - 难以映射到粗粒度的机器

Machine Model 2a: SIMD System

- 大量小处理器
  - 由一个“控制处理器”发布每条指令
  - 每个处理器执行相同的指令
  - 一些处理器可能在某些指令下被关闭

- 特点:
  - 为科学计算而设计
  - 不受厂商欢迎（CM2, Maspar）

- 编程模型可以编译实现
  - 映射n-fold并行到p处理器，n >> p，但在某些情况下（如HPF）很难
Machine Model 2b: Vector Machines

- Vector architectures are based on a single processor
  - Multiple functional units
  - All performing the same operation
  - Instructions may specify large amounts of parallelism (e.g., 64-way) but hardware executes only a subset in parallel
- Historically important
  - Overtaken by MPPs in the 90s
- Re-emerging in recent years
  - At a large scale in the Earth Simulator (NEC SX6) and Cray X1
  - At a small scale in SIMD media extensions to microprocessors
    - SSE, SSE2 (Intel: Pentium/IA64)
    - Altivec (IBM/Motorola/Apple: PowerPC)
    - VIS (Sun: Sparc)
- Key idea: Compiler does some of the difficult work of finding parallelism, so the hardware doesn’t have to

\[
\pi = \int_0^1 \frac{4}{1 + x^2} \, dx \approx \sum_{0 \leq i < N} \frac{4}{1 + \left(\frac{i + 0.5}{N}\right)^2} \cdot \frac{1}{N}
\]
### Programming Model 1: Shared Memory

- Program is a collection of threads of control.
- Can be created dynamically, mid-execution, in some languages.
- Each thread has a set of private variables, e.g., local stack variables.
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
- Threads communicate implicitly by writing and reading shared variables.
- Threads coordinate by synchronizing on shared variables.
- \( y = s \)...
Improved Code for Computing a Sum

Thread 1

\[
\begin{align*}
\text{local}_s1 &= 0 \\
\text{for } i &= 0, n/2-1 \\
\text{local}_s1 &= \text{local}_s1 + f(A[i]) \\
\text{lock}(\text{lk}); \\
s &= s + \text{local}_s1 \\
\text{unlock}(\text{lk}); \\
\end{align*}
\]

Thread 2

\[
\begin{align*}
\text{local}_s2 &= 0 \\
\text{for } i &= n/2, n-1 \\
\text{local}_s2 &= \text{local}_s2 + f(A[i]) \\
\text{lock}(\text{lk}); \\
s &= s + \text{local}_s2 \\
\text{unlock}(\text{lk}); \\
\end{align*}
\]

- Since addition is associative, it's OK to rearrange order
- Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - But there is still a race condition on the update of shared s
  - The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)

```c
#define N 1000000
main() { 
    double local, pi = 0.0, w;  
    long i;  
    w = 1.0/N;  
    #Pragma Parallel  
    #Pragma Shared(pi,w)  
    #Pragma Local(i,local)  
    {  
        #Pragma pfor iterate(i=0;N;1)  
        for (i = 0; i<N; i ++) {  
            local = (i + 0.5)*w;  
            local = 4.0/(1.0+local * local);  
        }  
        #Pragma Critical  
        pi = pi + local;  
    }  
    printf("pi is %f \n", pi *w);  
}
```

消息传递（Message Passing）

- 概况：
  - MPP, COW的自然模型
- 特点：
  - 多线程
  - 异步
  - 多地址空间
  - 显式同步
  - 显式数据映射和负载分配
  - 显式通信

Programming Model 2: Message Passing

- Program consists of a collection of named processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space -- NO shared data.
  - Logically shared data is partitioned over local processes.
- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI is the most common example

---

http://gr.xjtu.edu.cn/web/zhaoy

- First possible solution – what could go wrong?
  
  **Processor 1**
  
  \[
  \begin{align*}
  &xlocal = A[1] \\
  &send xlocal, proc2 \\
  &receive xremote, proc2 \\
  &s = xlocal + xremote
  \end{align*}
  \]

  **Processor 2**
  
  \[
  \begin{align*}
  &send xlocal, proc1 \\
  &receive xremote, proc1 \\
  &s = xlocal + xremote
  \end{align*}
  \]

- If send/receive acts like the telephone system? The post office?

  **Second possible solution**

  **Processor 1**
  
  \[
  \begin{align*}
  &xlocal = A[1] \\
  &send xlocal, proc2 \\
  &receive xremote, proc2 \\
  &s = xlocal + xremote
  \end{align*}
  \]

  **Processor 2**
  
  \[
  \begin{align*}
  &receive xremote, proc1 \\
  &send xlocal, proc1 \\
  &s = xlocal + xremote
  \end{align*}
  \]

MPI – the de facto standard

In 2002 MPI has become the de facto standard for parallel computing

The software challenge: overcoming the MPI barrier

- MPI created finally a standard for applications development in the HPC community
- Standards are always a barrier to further development
- The MPI standard is a least common denominator building on mid-80s technology

Programming Model reflects hardware!

“I am not sure how I will program a Petaflops computer, but I am sure that I will need MPI somewhere” – HDS 2001

---

```c
#define N 1000000
int MPI_Reduce ( void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )

main() {
    double local, pi, w;
    long i, taskid, numtask;
    w = 1.0/N;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Comm_Size(MPI_COMM_WORLD, &numtask);
    for (i = taskid; i < N; i = i + numtask) {
        local = (i + 0.5)*w;
        local = 4.0/(1.0 + local * local);
    }
    MPI_Reduce(&local, &pi, 1, MPI_Double, MPI_MAX, 0, MPI_COMM_WORLD);
    if (taskid == 0) printf("pi is %f \n", pi * w);
    MPI_Finalize();
}
```
Cilk

- Cilk is a language for multithreaded parallel programming based on ANSI C. Cilk is designed for general-purpose parallel programming, but it is especially effective for exploiting dynamic, highly asynchronous parallelism, which can be difficult to write in data-parallel or message-passing style. Using Cilk, our group has developed three world-class chess programs, StarTech, *Socrates, and Cilkchess. Cilk provides an effective platform for programming dense and sparse numerical algorithms, such as matrix factorization and N-body simulations, and we are working on other types of applications. Unlike many other multithreaded programming systems, Cilk is algorithmic, in that the runtime system employs a scheduler that allows the performance of programs to be estimated accurately based on abstract complexity measures.

The Cilk language has been developed since 1994 at the MIT Laboratory for Computer Science. 
http://supertech.csail.mit.edu/cilk/

Commercialization of Cilk Technology

- Prior to ~2006, the market for Cilk was restricted to high-performance computing. The emergence of multicore processors in mainstream computing means that hundreds of millions of new parallel computers are now being shipped every year. Cilk Arts was formed to capitalize on that opportunity: In 2006, Professor Leiserson launched Cilk Arts to create and bring to market a modern version of Cilk that supports the commercial needs of an upcoming generation of programmers. The company closed a Series A venture financing round in October 2007, and Cilk++ 1.0 shipped in December, 2008. Cilk++ differs from Cilk in several ways: support for C++, operation with both Microsoft and GCC compilers, support for loops, and "Cilk hyperobjects" - a new construct designed to solve data race problems created by parallel accesses to global variables.

Charles Eric Leiserson is a computer scientist, specializing in the theory of parallel computing and distributed computing.

WIKI定义

- Cilk is a general-purpose programming language designed for multithreaded parallel computing.
- The biggest principle behind the design of the Cilk language is that the programmer should be responsible for exposing the parallelism, identifying elements that can safely be executed in parallel; it should then be left to the run-time environment, particularly the scheduler, to decide during execution how to actually divide the work between processors. It is because these responsibilities are separated that a Cilk program can run without rewriting on any number of processors, including one.

Basic parallelism with Cilk

- **spawn** -- this keyword indicates that the procedure call it modifies can safely operate in parallel with other executing code. Note that the scheduler is not obligated to run this procedure in parallel; the keyword merely alerts the scheduler that it can do so.
- **sync** -- this keyword indicates that execution of the current procedure cannot proceed until all previously spawned procedures have completed and returned their results to the parent frame. This is an example of a barrier method.
### Introducing Clikk

```c
clik int fib(int n) {
    if (n < 2) return n;
    else {
        int n1, n2;
        n1 = spawn fib(n-1);
        n2 = spawn fib(n-2);
        sync;
        return (n1 + n2);
    }
}
```

- Clik constructs
  - `clik`: Clikk function. without it, functions are standard C
  - `spawn`: call can execute asynchronously in a concurrent thread
  - `sync`: current thread waits for all locally-spawmed functions
- Clik constructs specify logical parallelism in the program:
  - what computations can be performed in parallel
  - not mapping of tasks to processes

### Clikk Terminology

- **Parallel control** = `spawn`, `sync`, `return` from spawned function

- **Thread** = maximal sequence of instructions not containing parallel control (task in earlier terminology)

```c
clik int fib(n) {
    if (n < 2) return n;
    else {
        int n1, n2;
        n1 = spawn fib(n-1);
        n2 = spawn fib(n-2);
        sync;
        return (n1 + n2);
    }
}
```

### Clikk Language

- **Clik is a faithful extension of C**
  - If Clik keywords are elided -> C program semantics
- **Idiosyncrasies**
  - `spawn` keyword can only be applied to a `clik` function
  - `spawn` keyword cannot be used in a C function
  - `clik` function cannot be called with normal C call conventions
    - must be called with a `spawn` & waited for by `sync`
Cilk Program Execution as a DAG

- Each circle represents a thread.

Legend:
- continuation
- spawn
- return

Work and Critical Path Example

If all threads run in unit time:
- \( T_c = 17 \)
- \( T = 8 \) (critical path length)

Performance Measures

- \( T_1 \) = sequential work; minimum running time on 1 processor
- \( T_P \) = minimum running time on \( P \) processors
- \( T \) = minimum running time on infinite number of processors
  - equivalent to longest path in DAG \( \rightarrow \) critical path length

Properties of Performance Measures

- \( T_P \geq T \)
  - \( P \) processors can do at most \( P \) work in one step
  - suppose \( T_P < T \), then \( P T_P < T \) (a contradiction)
- \( T_P \geq T \)
  - suppose not: \( T_P < T \)
  - could use \( P \) of unlimited processors to reduce \( T \)
- \( T \) / \( T_P \) = speedup
  - with \( P \) processors, maximum speedup is \( P \) (for simplified model)
    - possibilities:
      - linear speedup: \( T_1 / T_P = \Theta(F) \)
      - sublinear speedup: \( T_1 / T_P = \alpha(P) \)
    - superlinear speedup: \( T_1 / T_P = \omega(P) \) (never with simplified model)
- \( T_1 / T \) = maximum speedup on \( \infty \) processors
The number of function invocations for such a function is exponential, that is: \( O(2^n) \).

The number of leaves, in general, is precisely \( \text{Fib}(n+1) \).

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots \]
### Cactus Stacks

- Cilk uses a cactus stack
- A cactus stack enables sharing of C function’s local variables

```c
void A() { B(); C(); }
void B() { D(); }
void C() { F(); }
void D() { }
void E() { }
void F() { }
```

- Each procedure’s view of stack:
  - A
  - B
  - C
  - D
  - E
  - F

#### Rules
- Pointers can be passed down call chain
- Only pass pointers up if they point to heap
- Functions cannot return pointers to local variables

### Race Conditions

- Two or more concurrent accesses to the same variable
- At least one is a write

```c
void g(int x) {
    serial semantics? parallel semantics?
    int x = 0;
    f returns 2
    let’s look closely
    spawn g(x);
    spawn g(x);
    sync;
    parallel execution of two instances of g: g, g
    return x;
}
```

**Example:**

- Call graph:
  - A
  - B
  - C
  - D
  - E
  - F

#### Rules
- One interleaving:
  - Read x
  - Add 1
  - Write x

### Greedy Scheduling

- Types of schedule steps
  - Complete step
    - at least P threads ready to run
    - No other remaining threads
  - Incomplete step
    - Strictly < P threads ready to run
    - Greedy scheduler runs them all

#### Theorem:
On P processors, a greedy scheduler executes any computation G with work T₁ and critical path length T in time T₂ ≤ T₁/P + T

**Proof sketch:**
- Only two types of scheduler steps: complete, incomplete
- Cannot be more than T₁/P complete steps, else work > T₁
- Every incomplete step reduces remaining critical path length by 1
- No more than T incomplete steps

### What’s the Problem with Races?

- Different interleavings can produce different results
- Race conditions cause non-deterministic behavior
  - Executions may not be repeatable
  - Multiple executions may yield different results
Programming with Race Conditions

- Approach 1: avoid them completely
  - no read/write sharing between concurrent tasks
  - only share between child and parent tasks in Cilk
- Approach 2: be careful!
  - sometimes, outcome of a race won’t affect overall result
    - e.g. processes sharing a work queue
  - the order in which processes grab tasks is immaterial to the result that the work gets performed
- avoiding data corruption
  - word operations are atomic on microprocessor architectures
  - definition of a word varies according to processor: 32-bit, 64-bit
  - use locks to control atomicity of aggregate structures
  - acquire lock
  - read and/or write protected data
  - release lock

Challenge Problem: N Queens

- Problem
  - place N queens on an N x N chess board
  - no 2 queens in same row, column, or diagonal
- Solution sketch

```cilk
void nqueens(int i, int j, int k, int n)
{
    if (k == n) {
        // place j-th queen in k-th position
        // if this is a legal placement of j-th queen
        // spawn nqueens(j+1,...)
        // sync;
        if (some child found a legal result return one, else return null)
    }
}
```

- An inefficiency
  - a single placement suffices; no need to compute all legal placements
  - so far, no way to terminate children exploring alternate placements

References

- Cilk 5.4.1 reference manual.
  [http://theory.lcs.mit.edu/classes/6.895/fall03/scribe/master.ps](http://theory.lcs.mit.edu/classes/6.895/fall03/scribe/master.ps)

abort

- Syntax: abort;
- Where: within a cilk procedure `p`
- Purpose: terminate execution of all of `p`’s spawned children
- Does this help with our nqueens example?

```cilk
void nqueens(int i, int j, int n)
{
    if (i == n) {
        // place j-th queen in n-th position
        // if this is a legal placement of j-th queen
        // spawn nqueens(n+1,...)
        // sync;
        if (some child found a legal result return one, else return null)
    }
}
```

Not yet! Need a way to invoke abort when a child yields a solution

http://gr.xjtu.edu.cn/web/zhaoy
inlet

- Normal spawn: `x = spawn f(...)`
  - result of `f` simply copied into caller's frame

- Problem
  - might want to handle receipt of a result immediately
  - queues handle legal placement returned from child promptly

- Solution: `inlet`
  - block of code within a function used to incorporate results
  - executes atomically with respect to enclosing function

- Syntax (`inlet` must appear in declarations section)

```plaintext
incl int f(int n) {
    ... (code using inlet) ...
    return;
}
```

Using an inlet

A simple complete example

```plaintext
incl int f(int n) {
    if (n < 2) return n;
    else {
        int n1, n2;
        n1 = spawn f(n-1); // spawn a child
        n2 = spawn f(n-2);
        sync;
        return (n1 + n2);
    }
}
```

N Queens Revisited

New solution that finishes when first legal result discovered

```plaintext
click void queens(int i, placement[])
    int 'result' = null;
    if (i == columns) return result; // base case
    for (placement[k] = 0; k < columns; k++)
        if (legal(placement, k))
            placement[k] = i;
            if (i == columns - 1) return result; // base case
            else {
                queens(i+1, placement);
                if (result != null) return result;
                placement[k] = 0;
            }
    }
```

Implicit inlets

- General `spawn` syntax
  - `spawn proc( arg1, ..., argn);`
  - `proc` may be omitted
  - `spawn update( data);`
  - `if` is present
      - it must be a variable matching the return type for the function
      - `op` may be
        - `=` `+=` `-=` `<<=` `>>=` `&=` `|=` `^=` `|=`

- Implicit inlets execute atomically w.r.t. caller
Using an implicit `inlet`

click int fib(int n) {
    if (n < 2) return n;
    int n1, n2;
    n1 = spawn fib(n-1);
    n2 = spawn fib(n-2);
    sync;
    return (n1 + n2);
}

Click guarantees implicit inlet instances
from all spawned children are atomic w.r.t.
on another and caller.

Locks

- Why locks? Guarantee mutual exclusion to shared state
  - only way to guarantee atomicity when concurrent procedure
    instances are operating on shared data

- Library primitives for locking
  `Clik_lock_init(Clik_lockvar k)`
  `Clik_lock(Clik_lockvar k)`
  `Clik_unlock(Clik_lockvar k)`

Concurrency Cautions

- Clik atomicity guarantees
  - all threads of a single procedure operate atomically
  - threads of a procedure include
    - all code in the procedure body proper, including inlet code

- Guarantee implications
  - can coordinate caller and callees using inlets without locks

- Only limited guarantees between descendants or ancestors
  - DAG precedence order maintained and nothing more
  - don’t assume atomicity between different procedures!
Sorting in Cilk: cilsort

Variant of merge sort
- Divide array into four quarters A, B, C, D of equal size
- Sort each quarter recursively in parallel
- merge sorted A & B into C and C & D into E (in parallel)
- merge sorted A & C into A

High-level sketch

cilk void cilsort(low, tap, size)
{
    if size <= 1 return input
    size = size/4
    spawn cilsort(A, ta, size/4); spawn cilsort(B, tB, size/4);
    spawn cilsort(C, tc, size/4); spawn cilsort(D, tD, size/4);
    sync;
    spawn cilsmerge(A, A + size/4, B, B + size/4, TA);
    spawn cilsmerge(C, C + size/4, D, D + size/4, TC);
    sync;
    spawn cilsmerge(tA, TA + size/4, tC, tC + size/4, A);
    sync;
}

Optimizing Performance of cilsort

- Recursively subdividing all the way to singletons is expensive
- When size(remaining sequence) to sort or merge is small (2K)
  — use sequential quicksort
  — use sequential merge
- Remaining issue: does not optimally use memory hierarchy
- Funnelsort is optimal in this regard
  — split input into n^m sections of size n^{m-1}
  — sort each recursively in parallel
  — merge n^{m-1} sorted sequences using an n^{m}/2-way merger
  — funnelsort(n): only O(ln(n)(l+log(n))) cache misses if z = \Omega(L^2)
  — See [Frigo MIT PhD 99]

Cilk: Behind the Curtain

- cilk generates two copies of each procedure
  — fast: for optimized execution on a single processor
  — slow: used to handle execution of "stolen procedure frames"
  — key support for Cilk's work-stealing scheduler
- Two schedulers
  — nanoscheduler: compiled into cilk program
    — execute cilk procedure and spawns in exactly the same order as C
    — on one PE: when no microscheduling needed, same order as C
    — efficient coordination with microscheduler
  — microscheduler
    — schedule procedures across a fixed set of processors
    — implementation: randomized work-stealing scheduler
      — when a processor runs out of work, it becomes a thief
      — steals from victim processor chosen uniformly at random

http://gr.xjtu.edu.cn/web/zhaoy
Microscheduler

Schedule procedures across a fixed set of processors

- When a processor runs out of work, it becomes a thief
  - steals from victim processor chosen uniformly at random
- When it finds victim with frames in its deque
  - takes the topmost frame (most recently pushed)
  - places frame into its own deque
  - gives the corresponding procedure to its own nanoscheduler
- Nanoscheduler executes slow version of the procedure
  - passes only pointer to frame as argument
    - real args and local state in frame
  - restores pgm counter to proper place using switch stmt (Duff's device)
  - at a sync, must wait for children
  - before the procedure returns, place return value into frame

Nanscheduler Sketch

- Upon entering a click function
  - allocate a frame in the heap
  - initialize the frame to hold the function’s shared state
  - push the frame into the bottom of a deque (doubly-ended queue)
  - one-to-one mapping between frames on stack and in deque
- At a spawn
  - save the state of the function into the frame
    - only live, dirty variables
  - save the entry number (position in the function) into the frame
  - call the spawned procedure with a normal function call
- After each spawn
  - check to see if the procedure has been migrated
    - if the current frame is still in the deque, then it has not
    - if so, clean up C stack
- Each sync becomes a no-op
- When the procedure returns
  - pop the frame off the deque
  - resume the caller after the spawn that called this procedure

Nanoscheduler Overheads

Basis for comparison: serial C

- Allocation and initialization of frame, push onto deque
  - a few assembly instructions
- Procedure’s state needs to be saved before each spawn
  - entry number, live variables
  - memory synchronization for non-sequentially consistent SMPs
- Check whether frame is stolen after each spawn
  - two reads, compare, branch (= memory synch if needed)
- On return, free frame - a few instructions
- One extra variable to hold frame pointer
- Overhead in practice
  - fib(n) runs ~ factor of 2 or 3 slower than seq C
References

- Cilk 5.4.1 reference manual.
  [http://theory.lcs.mit.edu/classes/6.895/fall03/scribe/master.ps](http://theory.lcs.mit.edu/classes/6.895/fall03/scribe/master.ps)