

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

并行程序设计方法

串行代码段

```
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];
```

(a) 使用库例程构造并行程序

```
id=my_process_id();
p=number_of_processes();
for ( i= id; i<N; i=i+p) A[i]=b[i]*b[i+1];
barrier();
for (i= id; i<N; i=i+p) c[i]=A[i]+A[i+1];
```

例子: MPI, PVM, Pthreads

(b) 扩展串行语言

my_process_id, number_of_processes(), and barrier()

A(0:N-1)=b(0:N-1)*b(1:N)

c=A(0:N-1)+A(1:N)

例子: Fortran 90

(c) 加编译制导构造并行程序的方法

```
#pragma parallel
#pragma shared(A,b,c)
#pragma local(i)
{
# pragma pfor iterate(i=0;N;1)
for (i=0;i<N;i++) A[i]=b[i]*b[i+1];
# pragma synchronize
# pragma pfor iterate (i=0; N; 1)
for (i=0;i<N;i++)c[i]=A[i]+A[i+1];
}
```

例子: SGI power C

(d) 自动并行: Autopar

(e) 新的并行语言: Linda, Ocean

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

并行编程风范

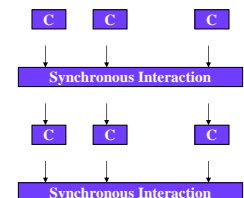
- ❖ 相并行 (Phase Parallel)
- ❖ 分治并行 (Divide and Conquer Parallel)
- ❖ 流水线并行 (Pipeline Parallel)
- ❖ 主从并行 (Master-Slave Parallel)
- ❖ 工作池并行 (Work Pool Parallel)

并行程序设计环境与工具

- ❖ 并行程序设计过程
 - ⊕ 应用问题的具体算法
 - ⊕ 在并行计算模型上编程实现算法
 - ⊕ 编译器生成目标代码
 - ⊕ 借助操作系统和硬件平台运行程序
- ❖ 整个并行程序设计过程中, 编程环境与编程工具起着重要作用
 - ⊕ 编程环境: 包括硬件平台、支撑语言、操作系统、软件工具等
 - ⊕ 编程工具: 帮助用户开发应用问题的软件工具
 - 作业管理工具、查错工具、性能分析工具等
 - ⊕ 集成工具

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



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:
 - ✦ 多线程
 - ✦ 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

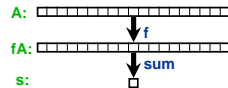
Machine Model 2b: Vector Machines

- ❖ 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

Programming Model 2: Data Parallel

- ❖ Single thread of control consisting of parallel operations.
- ❖ Parallel operations applied to all (or a defined subset) of a data structure, usually an array
 - ✦ Communication is implicit in parallel operators
 - ✦ Elegant and easy to understand and reason about
 - ✦ Coordination is implicit – statements executed synchronously
- ❖ Drawbacks:
 - ✦ Not all problems fit this model
 - ✦ Difficult to map onto coarse-grained machines

A = array of all data
 $fA = f(A)$
 $s = \text{sum}(fA)$

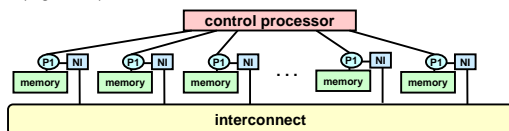


Vector Processors

- ❖ Vector instructions operate on a vector of elements
 - ✦ These are specified as operations on vector registers
-
- The diagram shows three horizontal boxes representing vector registers, labeled vr1, vr2, and vr3. Arrows from each register point to a central point, with a note: "(logically, performs # elts adds in parallel)".
- ❖ A supercomputer vector register holds ~32-64 elts
 - ✦ The number of elements is larger than the amount of parallel hardware, called vector pipes or lanes, say 2-4
 - ❖ The hardware performs a full vector operation in
 - ✦ #elements-per-vector-register / #pipes
-
- The diagram shows multiple horizontal boxes for vector registers (vr1, vr2, etc.) and multiple circular nodes representing pipes. Arrows connect the registers to the pipes, with a note: "(actually, performs # pipes adds in parallel)".

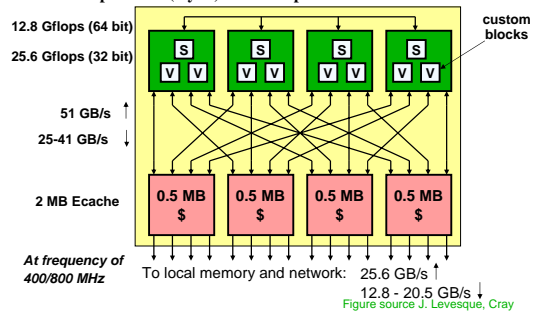
Machine Model 2a: SIMD System

- ❖ A large number of (usually) small processors.
 - ✦ A single "control processor" issues each instruction.
 - ✦ Each processor executes the same instruction.
 - ✦ Some processors may be turned off on some instructions.
- ❖ Machines are very specialized to scientific computing, so they are not popular with vendors (CM2, Maspar)
- ❖ Programming model can be implemented in the compiler
 - ✦ mapping n-fold parallelism to p processors, $n \gg p$, but it's hard (e.g., HPF)

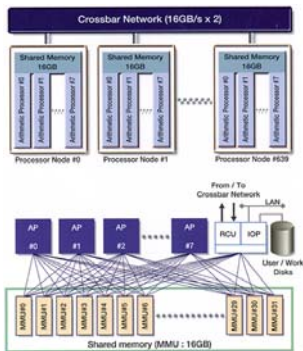


Cray X1 Node

- ❖ Cray X1 builds a larger "virtual vector", called an MSP
 - ✦ 4 SSPs (each a 2-pipe vector processor) make up an MSP
 - ✦ Compiler will (try to) vectorize/parallelize across the MSP



Earth Simulator Architecture



Parallel Vector Architecture

- High speed (vector) processors
- High memory bandwidth (vector architecture)
- Fast network (new crossbar switch)

Rearranging commodity parts can't match this performance

```
main() {
    long i,j,t,N=1000000;
    double local[N],tmp[N], pi, w;
    w=1.0/N;
    forall (i = 0; i<N; i++) {
        local[i] = (i + 0.5)*w;
        tmp[i] = tmp[i] + 4.0/(1.0+local[i] * local[i]);
    }
    pi=sum(tmp);
    printf("pi is %f \n", pi *w);
}
```

计算圆周率的样本程序

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \approx \sum_{0 \leq i < N} \frac{4}{1 + \left(\frac{i+0.5}{N}\right)^2} \cdot \frac{1}{N}$$

共享变量 (Shared Variable)

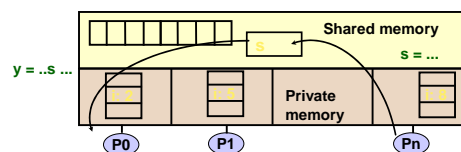
- ❖ 概况:
 - PVP, SMP, DSM的自然模型
- ❖ 特点:
 - 多线程: SPMD, MPMD
 - 异步
 - 单一地址空间
 - 显式同步
 - 隐式数据分布
 - 隐式通信

计算圆周率的c语言代码段

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

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



Shared Memory Code for Computing a Sum

```

static int s = 0;

Thread 1
for i = 0, n/2-1
s = s + f(A[i])

Thread 2
for i = n/2, n-1
s = s + f(A[i])
    
```

- Problem is a race condition on variable s in the program
- A **race condition** or **data race** occurs when:
 - two processors (or two threads) access the same variable, and at least one does a write.
 - The accesses are concurrent (not synchronized) so they could happen simultaneously

```

#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);
}
    
```

Shared Memory Code for Computing a Sum

```

static int s = 0;

Thread 1
...
compute f(A[i]) and put in reg0 7
reg1 = s 27
reg1 = reg1 + reg0 34
s = reg1 34
...

Thread 2
...
compute f(A[i]) and put in reg0 9
reg1 = s 27
reg1 = reg1 + reg0 36
s = reg1 36
...
    
```

- Assume s=27, f(A[i])=7 on Thread1 and =9 on Thread2
- For this program to work, s should be 43 at the end
 - but it may be 43, 34, or 36
- The atomic operations are reads and writes
 - Never see 1/2 of one number
 - All computations happen in (private) registers

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

Improved Code for Computing a Sum

```

static int s = 0;
static lock lk;

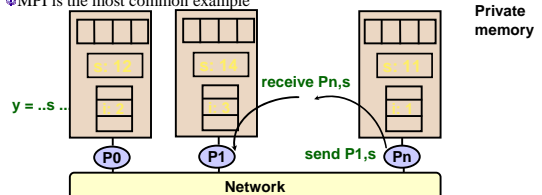
Thread 1
local_s1 = 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
lock(lk);
s = s + local_s1
unlock(lk);

Thread 2
local_s2 = 0
for i = n/2, n-1
    local_s2 = local_s2 + f(A[i])
lock(lk);
s = s + local_s2
unlock(lk);
    
```

- 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)

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



Computing $s = A[1] + A[2]$ on each processor

- First possible solution – what could go wrong?

```
Processor 1
xlocal = A[1]
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote
```

```
Processor 2
xlocal = A[2]
send xlocal, proc1
receive xremote, proc1
s = xlocal + xremote
```

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

- Second possible solution

```
Processor 1
xlocal = A[1]
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote
```

```
Processor 2
xlocal = A[2]
receive xremote, proc1
send xlocal, proc1
s = xlocal + xremote
```

共享存储并行编程

- ❖ Cilk
- ❖ Pthreads
- ❖ OpenMP

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

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/>

```
#define N 1000000
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();
}
```

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.

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

Cilk Language

- Cilk is a **faithful** extension of C
 - if Cilk keywords are elided → C program semantics
- Idiosyncrasies
 - spawn** keyword can only be applied to a **cilk** function
 - spawn** keyword cannot be used in a C function
 - cilk** function cannot be called with normal C call conventions
 - must be called with a **spawn** & waited for by a **sync**

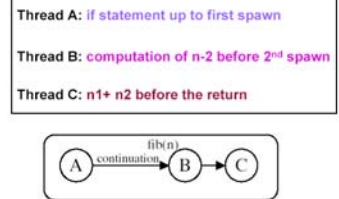
Basic parallelism with Cilk

- ❖ **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.

Cilk Terminology

- Parallel control = **spawn, sync, return** from spawned function
- Thread = maximal sequence of instructions not containing parallel control (task in earlier terminology)

```
cilk 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);
  }
}
```



Introducing Cilk

```
cilk 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);
  }
}
```

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

- ❖ 所谓continuation, 其实本来是一个函数调用机制。我们熟悉的函数调用方法都是使用堆栈, 采用Activation record或者叫Stack frame来记录从最顶层函数到当前函数的所有context。一个frame/record就是一个函数的局部上下文信息, 包括所有的局部变量的值和SP, PC指针的值(通过静态分析, 某些局部变量的信息是不必保存的, 特殊的如尾调用的情况则不需要任何stack frame。不过, 逻辑上, 我们认为所有信息都被保存了)。函数的调用前往往往伴随着一些push来保存context信息, 函数退出时则是取消当前的record/frame, 恢复上一个调用者的record/frame。象pascal这样的支持嵌套函数的, 则需要一个额外的指针来保存父函数的frame地址。不过, 无论如何, 在任何时候, 系统保存的就是一个后入先出的堆栈, 一个函数一旦退出, 它的frame就被删除了。Continuation则是另一种函数调用方式, 它不采用堆栈来保存上下文, 而是把这些信息保存在continuation record中。这些continuation record和堆栈的activation record的区别在于, 它不采用后入先出的线性方式, 所有record被组成一棵树(或者图), 从一个函数调用另一个函数就等于给当前节点生成一个子节点, 然后把系统寄存器移动到这个子节点。一个函数的退出等于从当前节点返回到父节点。这些节点的删除是由garbage collection来管理。如果没有引用这个record, 则它就可以被删除的。这样的调用方式和堆栈方式相比的好处在哪里呢? 最大的好处就是, 它可以让你从任意一个节点跳到另一个节点, 而不必遵循堆栈方式的一层一层的return方式。比如说, 在当前的函数内, 你只需要有一个其它函数的节点信息, 完全可以选择return到那个函数, 而不是循规蹈矩地返回到自己的调用者。你也可以在一个函数的任何位置储存自己的上下文信息, 然后, 在以后某个适当的时刻, 从其它的任何函数里面返回到自己现在的位置。

Scheduling Tasks in Cilk

- Alternative strategies
 - work-sharing**: thread scheduled to run in parallel at every spawn
 - benefit: maximizes parallelism
 - drawback: cost of setting up new threads is high → should be avoided
 - work-stealing**: processor looks for work when it becomes idle
 - lazy parallelism: put off work for parallel execution until necessary
 - benefits: executes with precisely as much parallelism as needed
 - minimizes the number of threads that must be set up
 - runs with same efficiency as serial program on uniprocessor
- Cilk uses **work-stealing** rather than **work-sharing**

Greedy Scheduling

- Types of schedule steps
 - complete step
 - at least P threads ready to run
 - select any P and run them
 - incomplete step
 - strictly $< P$ threads ready to run
 - greedy scheduler runs them all

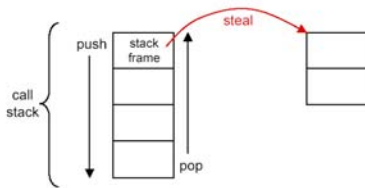
Theorem: On P processors, a greedy scheduler executes any computation G with work T_1 and critical path of length T_∞ in time $T_p \leq T_1/P + T_\infty$

Proof sketch

- only two types of scheduler steps: complete, incomplete
- cannot be more than T_1/P complete steps, else work $> T_1$
- every incomplete step reduces remaining critical path length by 1
 - no more than T_∞ incomplete steps

15

Call Stack of Executing Process



- Stack grows downward
- Stack frame contains local variables for a procedure invocation
- Procedure call → new frame is pushed onto the bottom of the stack
- Procedure return → bottom frame is popped from the stack
- Stack maintains order (synchronizes) between caller and callee

13

Race Conditions

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

```

cilk int f() {
  int x = 0;
  spawn g(&x);
  spawn g(&x);
  sync;
  return x;
}
    
```

serial semantics? f returns 2 parallel semantics? let's look closely

parallel execution of two instances of g: g, g
many interleavings possible

one interleaving

```

cilk void g(int *p)
{
  *p += 1;
}
    
```

read x
add 1
write x

read x
read x
add 1
add 1
write x
write x

f returns 1!

16

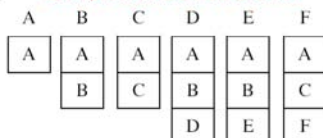
Cactus Stacks

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

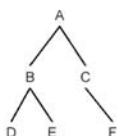
```

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

each procedure's view of stack



call tree



Rules

- pointers can be passed down call chain
- only pass pointers up if they point to heap
 - functions cannot return ptrs to local variables

14

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

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(n,j, placement) {
  // precondition: placed j queens so far
  if (j == n) return placement
  for (k = 0; k < n; k++)
    place j+1 queen in kth position
    if this is a legal placement of j+1 queens
      spawn nqueens(n,j+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

4

References

- Cilk 5.4.1 reference manual.
- Charles Leiserson, Bradley Kuzmaul, Michael Bender, and Hua-wen Jing. MIT 6.895 lecture notes - Theory of Parallel Systems. <http://theory.lcs.mit.edu/classes/6.895/fall03/scribe/master.ps>

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
 - nqueens: 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)**

```
cilk int f(...) {
  inlet void my_inlet(ResultType* result, iarg2, ..., iargn) {
    // atomically incorporate result into f's variables
    return;
  }
  my_inlet(spawn g(...), iarg2, ..., iargn);
}
```

5

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(n,j,placement) {
  // precondition: placed j queens so far
  if (j == n) return placement
  for (k = 0; k < n; k++)
    place j+1 queen in kth position
    if this is a legal placement of j+1 queens
      spawn nqueens(n,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

3

Using an inlet

A simple complete example

```
cilk 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);
  }
}
```

cilk guarantees inlet instances from all spawned children are atomic w.r.t one another and caller too

```
cilk int fib(int n) {
  int result = 0;
  inlet void add(int r) {
    result += r;
    return;
  }
  if (n < 2) return n;
  else {
    int n1, n2;
    add(spawn fib(n-1));
    add(spawn fib(n-2));
    sync;
    return result;
  }
}
```

inlet has access to fib's variables

N Queens Revisited

New solution that finishes when first legal result discovered

```
cilk void nqueens(n,j,placement) {
  int *result = null;
  //precondition: placed j queens so far
  inlet void dorsull(chldplacement) {
    if (chldplacement == null) return, else { result = chldplacement; abort; }
  }
  if (j == n) return placement
  for (k = 0, k < n; k++)
    place j+1 queen in kth position
    if this yields a legal placement of j+1 queens
      dorsull(spawn nqueens(n,j+1,...))
  sync
  return result
}
```

function initializes result

if solution found, inlet updates result and aborts siblings

SYNCHED

- Determine whether a procedure has any currently outstanding children without executing `sync`
 - if children have not completed


```
SYNCHED = 0
```
 - otherwise


```
SYNCHED = 1
```
- Why **SYNCHED**? Save storage and enhance locality.

```
state1 = Cilk_alloc(state_size);
spawn foo(state1); /* fill in state1 with data */
if (SYNCHED) state2 = state1;
else state2 = Cilk_alloc(state_size);
spawn bar(state2);
sync;
```

Implicit inlets

- General `spawn` syntax
 - statement: `[lhs op] spawn proc(arg1, ..., argn);`
 - `[lhs op]` may be omitted
 - `spawn update(&data);`
 - if `lhs` 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

implicit inlets

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

```
Cilk_lock_init(Cilk_lockvar k);
Cilk_lock(Cilk_lockvar k);
Cilk_unlock(Cilk_lockvar k);
```

must initialize a lock variable before using it!

Using an implicit inlet

```
cilk 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);
  }
}

cilk int fib(int n) {
  int result = 0;
  if (n < 2) return n;
  else {
    int n1, n2;
    result += spawn fib(n-1);
    result += spawn fib(n-2);
    sync;
    return result;
  }
}
```

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

Concurrency Cautions

- Cilk 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: cilk_sort

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 tA and C & D into tC (in parallel)
- merge sorted tA and tC into A

High-level sketch

```
cilk void cilk_sort(low,tmp,size){
    size4 = size/4
    if size <= 1 return input
    spawn cilk_sort(A,tA,size4); spawn cilk_sort(B,tB, size4);
    spawn cilk_sort(C, tC, size4);
    spawn cilk_sort(D, tD, size-3*size4);
    sync;
    spawn cilkmerge(A, A + size4-1, B, B + size4-1, tA);
    spawn cilkmerge(C, C + size4-1, D,low + size-1, tC);
    sync;
    spawn cilkmerge(tA, tC-1, tC, tA + size-1, A);
    sync;
}
```

13

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

Merging in Parallel

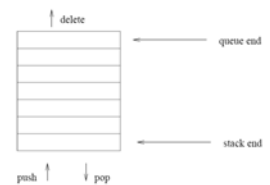
- How can you incorporate parallelism into a merge operation?
- Assume we are merging two sorted sequences A and B
- Without loss of generality, assume A larger than B

Algorithm Sketch

1. Find median of the elements in A and B (considered together).
2. Do binary search in A and B to find its position. Split A and B at this place to form A₁, A₂, B₁, and B₂
3. In parallel, recursively merge A₁ with B₁ and A₂ with B₂

In the Cilk scheduling algorithm, a processor works on subroutine α until:

1. α spawns subroutine β
 - In this case, the processor pushes α to the bottom of the ready deque, and starts work on subroutine β .
2. α returns
 - If the deque is nonempty, the processor pops the bottom subroutine and begins working on it.
 - If the deque is empty, first the processor tries to execute α 's parent.
 - If α 's parent is busy, the processor steals work at random.
3. α syncs with another subroutine
 - If there exists outstanding children and the computation cannot proceed, then the processor worksteals. Note that the deque must be empty in this case.



- The processor chooses a victim uniformly at random.
- If the victim's deque is empty, the processor tries again.
- Otherwise, the processor steals the top (oldest) thread of the victim and begins to work on it.

Optimizing Performance of cilk_sort

- 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 $n^{1/3}$ sections of size $n^{2/3}$
 - sort each recursively in parallel
 - merge $n^{1/3}$ sorted sequences using an $n^{1/3}$ -way merger
 - funnel sort(n): only $O(1+(n/L)(1+\log_n))$ cache misses if $z = \Omega(L^2)$
- See [Frigo MIT PhD 99]

Nanscheduler Sketch

- Upon entering a cilk 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 pairing 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

17

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 (least recently pushed)
 - places frame into its own deque
 - gives the corresponding procedure to its own nanoscheduler
- Nanoscheduler executes **slow** version of the **procedure**
 - receives only pointer to frame as argument Using slow procedure
 - 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

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.
- Matteo Frigo. Portable High-performance Programs. PhD thesis. MIT, 1999.
- Charles Leiserson, Bradley Kuszmaul, Michael Bender, and Hua-wen Jing. MIT 6.895 lecture notes - Theory of Parallel Systems.
<http://theory.lcs.mit.edu/classes/6.895/fall03/scribe/master.ps>