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硬件工具
    - 作业管理工具、查错工具、性能分析工具等
  - 集成工具

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

构造并行程序的途径

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) 自动并行化：Autopar

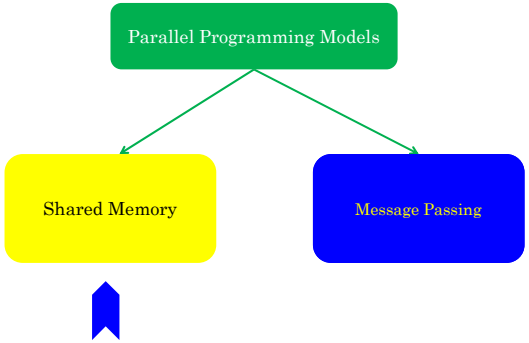
(e) 设计新并行语言：Linda, Ocean

并行编程模型

- What is a parallel programming model?
  - 是由硬件提供给程序员的一种抽象
  - 模型决定程序员怎样容易地将算法指定给并行计算单元（即，任务），这些单元为硬件所理解
  - 模型决定并行任务如何有效地在硬件上执行
- Main Goal:
  - 利用机器(例如, SMP, MPP, NUMA)的所有处理器；
  - 最小化程序的执行时间

5

传统编程模型



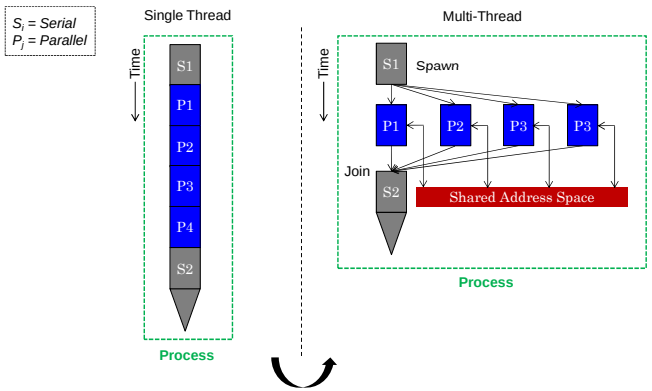
6

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 shared memory programming model

7

Shared Memory Model



8

Shared Memory Example

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
sum = 0;
for (i=0; i<8; i++)
  if (a[i] > 0)
    sum = sum + a[i];
Print sum;
```

Sequential

```
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[];
shared lock_type mylock;
start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];
barrier;
for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0) {
    lock(mylock);
      sum = sum + a[i];
    unlock(mylock);  }
barrier;   // necessary
end parallel // kill the child thread
Print sum;
```

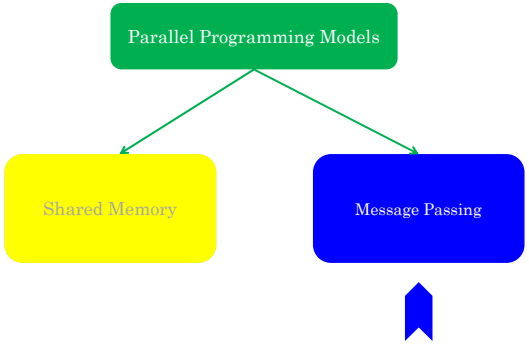
Parallel

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**

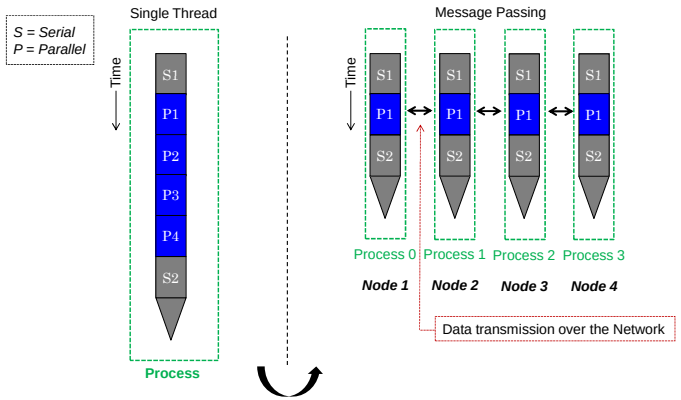
11

传统编程模型



10

Message Passing Model



12

Message Passing Example

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
sum = 0;
for (i=0; i<8; i++)
  if (a[i] > 0)
    sum = sum + a[i];
Print sum;
```

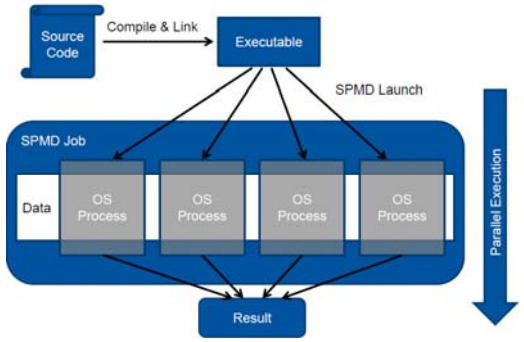
Sequential

```
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];
if (id == 0) {
  recv_msg (P1, &local_sum1);
  sum = local_sum + local_sum1;
  Print sum;
}else
  send_msg (P0, local_sum);
```

Parallel

并行编程例子：MPI SPMD/MPMD

- 基于消息传递的多个进程并行执行，是否需要程序级协调
- MPI主要机制：消息传递；笛卡尔拓扑；成组通信等



Shared Memory Vs. Message Passing

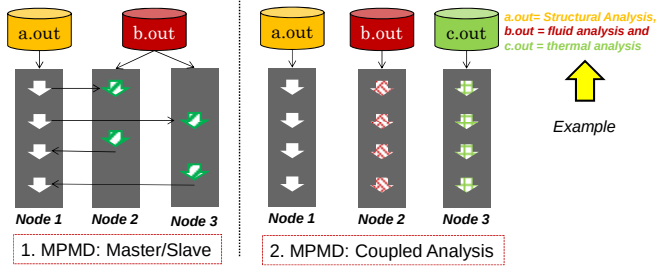
- 共享存储与消息传递编程模型比较:

Aspect	Shared Memory	Message Passing
Communication	**Implicit (via loads/stores)**	**Explicit Messages**
Synchronization	**Explicit**	**Implicit (Via Messages)**
Hardware Support	**Typically Required**	**None**
Development Effort	**Lower**	**Higher**
Tuning Effort	**Higher**	**Lower**

14

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



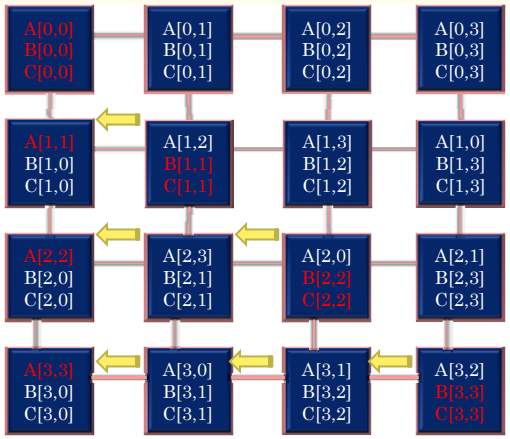
例：Cannon矩阵乘法

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

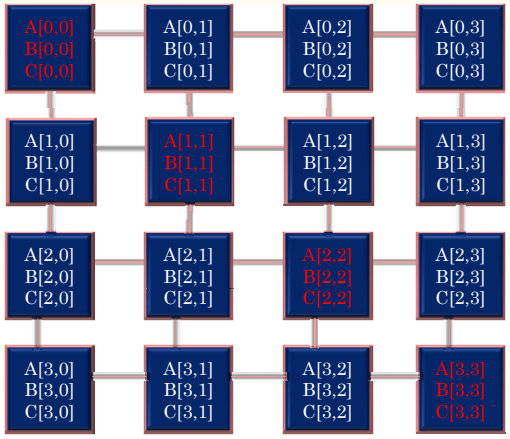
初始化 → 对齐 → 分块乘 → 重复 → 结果

- 移位
- 分块乘

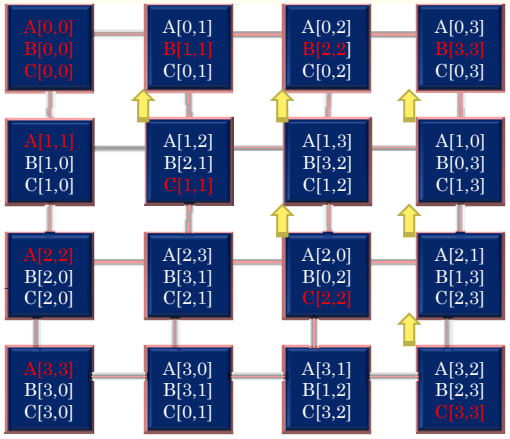
Cannon乘法：对齐A



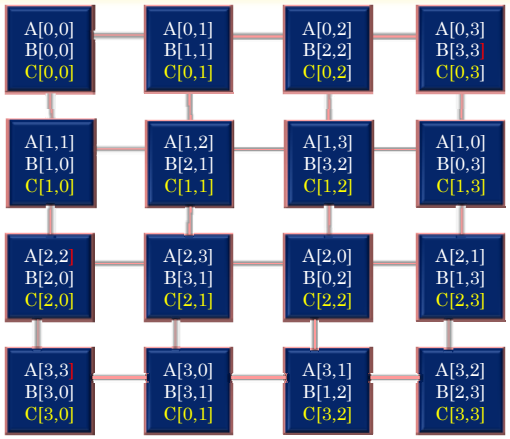
Cannon乘法：初始化



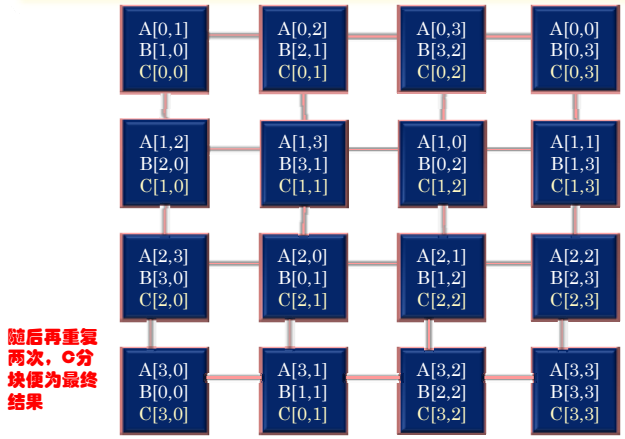
Cannon乘法：对齐B



每个结点并行乘分块得到C分块

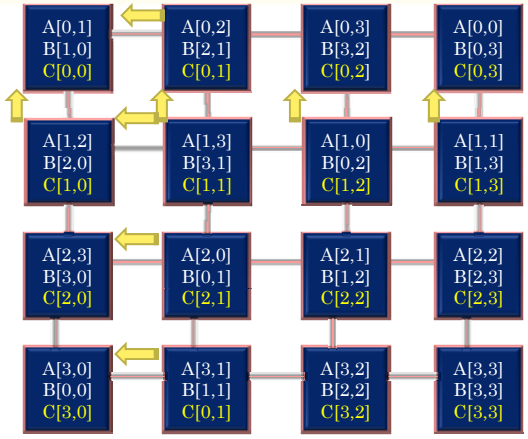


各自乘分块并累加到C分块



随后再重复两次，C分块便为最终结果

A和B各自循环移位



Cannon矩阵乘法MPI程序

- 参数a, b,和c 分别指向局部存储的分块，分别对应*A*, *B*, 和*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);
  /* Set the periods for wraparound connections */
  periods[0] = periods[1] = 1;
  /* Create the Cartesian topology, with rank reordering */
  MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
```

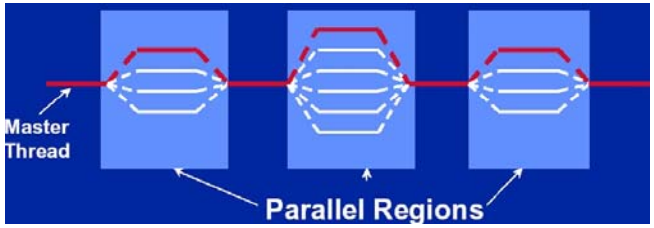
```
  /* Get into the main computation loop */
  for (i=0; i<dims[0]; i++) {
    MatrixMultiply(nlocal, a, b, c); /*c=c+a*b*/

  /* Compute ranks of the up and left shifts */
  MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
  MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
    /* Shift matrix a left by one */
    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
        leftrank, 1, rightrank, 1, comm_2d, &status);
    /* Shift matrix b up by one */
    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
        uprank, 1, downrank, 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);
```

```
  /* 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);
```

```
  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 */
}

/* 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];
}
```

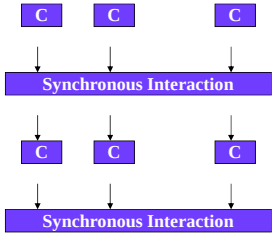
并行编程范型：OpenMP Master-Worker

- 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



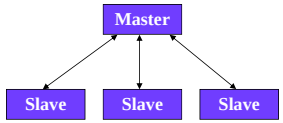
并行编程范型

- 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

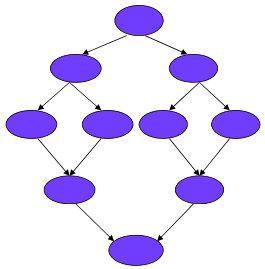
主一从并行（Master-Slave Parallel）

- 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



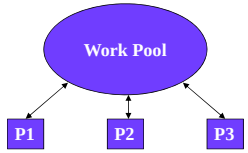
分治并行（Divide and Conquer Parallel）

- 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



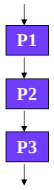
工作池并行（Work Pool Parallel）

- 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
- 直至任务池为空
- 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）

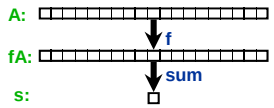
隐式并行（Implicit Parallel）

- 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

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 = sum(fA)

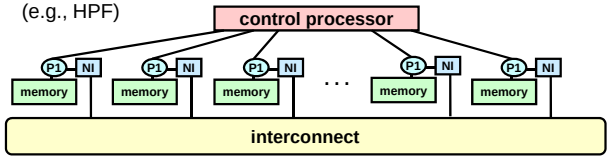


数据并行（Data Parallel）

- 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

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 >> p, but it's hard (e.g., HPF)



Machine Model 2b: Vector Machines

- **Vector architectures are based on a single processor**
  - Multiple functional units
  - All performing the same operation
  - Instructions may specific 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 sale 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**

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);
}
```

计算圆周率的样本程序

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

```
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);
}
```

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

Shared Memory Code for Computing a Sum

```
static int s = 0;
```

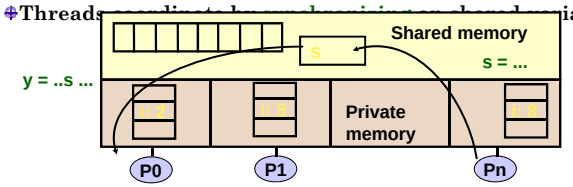
```
Thread 1

  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

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		Thread 2	
....		...	
compute f([A[i]) and put in reg0	7	compute f([A[i]) and put in reg0	9
reg1 = s	27	reg1 = s	27
reg1 = reg1 + reg0	34	reg1 = reg1 + reg0	36
s = reg1	34	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 ½ of one number
  - All computations happen in (private) registers

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)

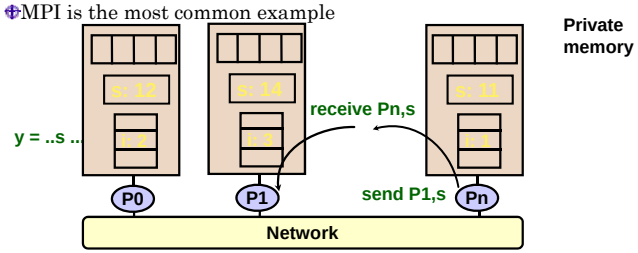
消息传递（Message Passing）

- 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

```
#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);
}
```

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

```
int MPI_Reduce ( void *sendbuf,
  void *recvbuf, int count,
  MPI_Datatype datatype,
  MPI_Op op, int root, MPI_Comm comm )
```

```
#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();
}
```

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

共享存储并行编程具体介绍

- **Cilk**
- **Pthreads**
- **OpenMP**

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 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

Cilk Terminology

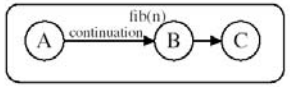
- Parallel control = spawn, sync, return from spawned function
- Thread = maximal sequence of instructions not containing parallel control (task in earlier terminology)

```
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);
 }
}
```

Thread A: if statement up to first spawn

Thread B: computation of n-2 before 2nd spawn

Thread C: n1+ n2 before the return

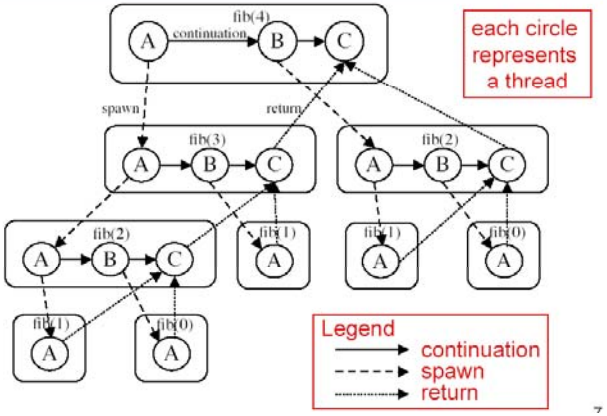


Cilk Language

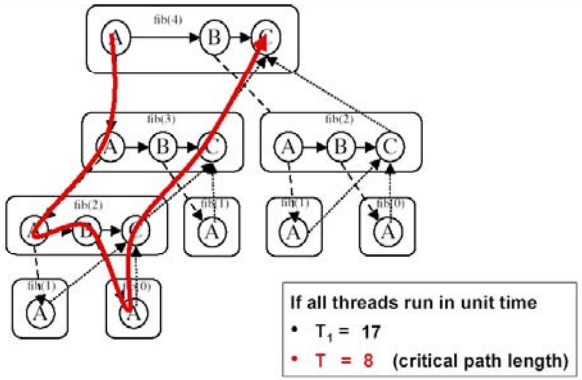
- Cilk is a faithful extension of C
  - —if Cilk keywords are elided → C program semantics
- Idiosyncrasies
  - —spawn keyword can only be applied to a cilk function
  - —spawn keyword cannot be used in a C function
  - —cilk function cannot be called with normal C call conventions
    - – must be called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

- 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退回到父节点。
  这些节点的删除是由garbage collection来管理。如果没有引用这个record，则它就是可以被删除的。
  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，从其它的任何一个函数里面返回到自己现在的位置。

Cilk Program Execution as a DAG



Work and Critical Path Example



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 → critical path length

Properties of Performance Measures

- $T_p \geq T_1/P$
  —P processors can do at most P work in one step
  —suppose $T_p < T_1/P$, then $PT_p < T_1$ (a contradiction)
- $T_p \geq T$
  —suppose not: $T_p < T$
  —could use P of unlimited processors to reduce T
- T_1/T_p = speedup
  —with P processors, maximum speedup is P (for simplified model)
  —possibilities
  – linear speedup: $T_1/T_p = \Theta(P)$
  – sublinear speedup: $T_1/T_p = o(P)$
  – superlinear speedup: $T_1/T_p = \Omega(P)$ (never with simplified model)
- T_1/T = maximum speedup on ∞ processors

Analyzing Parallelism in 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 int fib(int n) {
 if (n < 2) return n;
 else {
  int n1, n2;
  n1 = spawn fib(n-1);
  n2 = serial_fib(n-2);
  sync;
  return (n1 + n2);
 }
}
```

How much avg parallelism do we expect in each case?

$O(2^n/n)$ 2

What is the length of the critical path in each case? (counting operations, not threads)

$O(n)$ $O(2^n)$

What do we expect for parallel execution time?

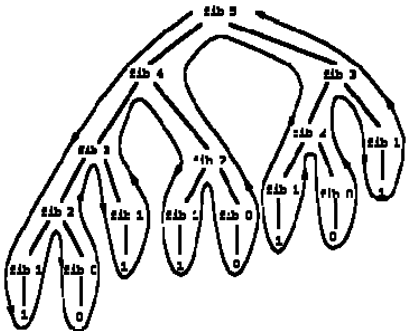
$O(2^n/P+n)$ $O(2^n)$

11

Ron Cytron, *et.al.* Automatic Generation of DAG Parallelism. ACM PLDI'89

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

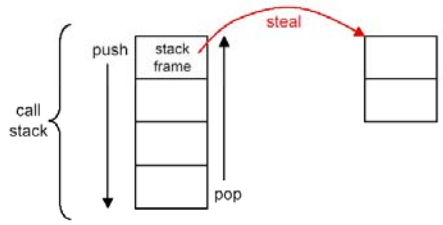


The number of function invocations for such a function is exponential, that is: $O(2^n)$

The number of leaves, in general, is precisely *Fib (n+1).*

0,1,1,2,3,5,8,13,21,...

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

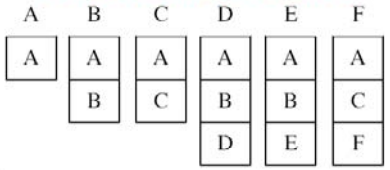
13

Cactus Stacks

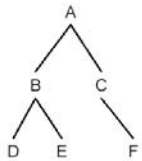
- Cilk uses a cactus stack
- A cactus stack enables sharing of C function's local variables

```
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

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 $T_p \le T_1/P + T$

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

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;
}

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

serial semantics? f returns 2

parallel semantics? let's look closely

parallel execution of two instances of g: g, g

many interleavings possible

read x
add 1
write x

one interleaving

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

f returns 1!

16

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(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

结束Cilk！

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

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

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

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 doresult(childplacement) {
    if (childplacement == null) return; else { result = childplacement; abort; }
  }
  if (j == n) return placement
  for (k = 0; k < n; k++)
    place j+1 queen in k^th position
    if this yields a legal placement of j+1 queens
      doresult(spawn nqueens(n,j+1,...))
  sync
  return result
}
```

function initializes result

if solution found, inlet updates result and aborts siblings

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 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;
 }
}
```

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

inlet has access to fib's variables

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

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

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!

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

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: cilksort

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 cilksort(low,tmp,size){
  size4 = size/4
  if size <= 1 return input
  spawn cilksort(A,tA,size4); spawn cilksort(B,tB, size4);
  spawn cilksort(C, tC, size4);
  spawn cilksort(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

Optimizing Performance of cilksort

- 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
  - —funnelsort(n): only $O(1+(n/L)(1+\log_z n))$ cache misses if $z = \Omega(L^2)$
    - See [Frigo MIT PhD 99]

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

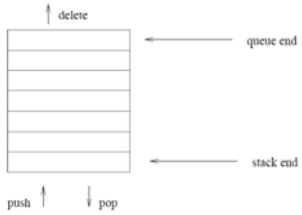
1. Find median of the elements in A and B (considered together).
2. Do binary search in A and B to find its position. Split A and B at this place to form A_1, A_2, B_1, and B_2
3. In parallel, recursively merge A_1 with B_1 and A_2 with B_2

Cilk: Behind the Curtain

- cilkc 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

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 *synchs* 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.

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
    - 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

Using slow procedure

Nanscheduler Sketch

Using fast procedure

- 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 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

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