MOCAA M^3 Workshop

Scheduling Parallel Algorithms in Computer Algebra

Marc Moreno Maza & Yuzhen Xie

What is this Talk About and not About?

- This talk is about:
  - an automatic scheduling software tool, Cilk
  - especially good for programs extracting parallelism from divide-and-conquer algorithms
  - which is particularly well-suited for symbolic computation,
  - although we will use numerical examples ...

- This talk is not about:
  - automatic scheduling in general,
  - programming environments for parallel symbolic computation,
  - why parallelism is good and important ...
- However, we will have a quick review ...

A Crash Course on Parallel Computing

- Measures of parallel efficiency:
  - for algorithms: running in $\log^{\alpha}(n)$ on $p = n^{O(1)}$ processors.
  - for programs: reaching linear speed-up (since p is small in practice).
- Reasons for parallel inefficiency:
  - no opportunities for concurrent execution,
  - unequal size tasks,
  - memory traffic which can limit the benefits of parallel execution
  - parallelism overheads:
    * cost of starting a thread or process,
    * cost of communicating shared data,
    * cost of synchronizing.
- Next we emphasize memory traffic issues ...

Efficient Implementation of Matrix Multiplication

• We aim at multiplying large square matrices of order n with floating point number coefficients. (Same story over a finite field.)

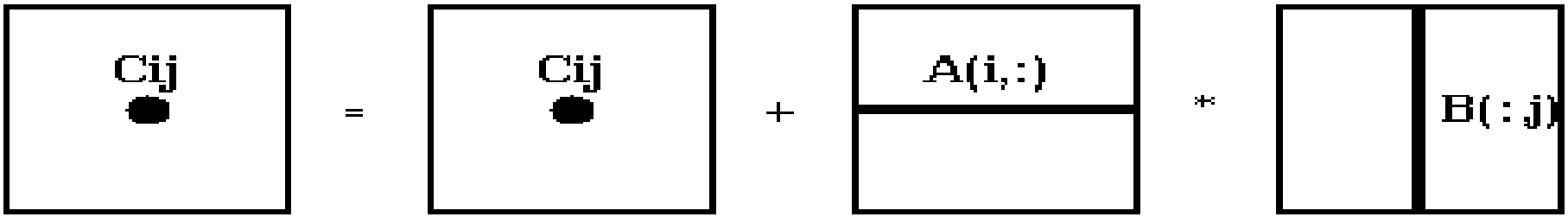
• Assumptions:

- Two levels of memory: slow and fast.
- The fast memory has size M words with $M << n^2$ but $M \geq 4n$.
- The input and output matrices reside entirely in slow memory.
- Each word is read from slow memory individually and we have full control over which words are transferred between the two levels.
- Finally, assume that one read/write access to slow memory is much more expensive than one operation on floating point numbers.

• Notations:

- Let m be the number of read/write access to slow memory
- Let $f = 2n^3$ the total number of operations on floating point numbers.

Unblocked Matrix Multiplication




```
for i=1 to n
   {Read row i of A into fast memory}
   for j=1 to n
      {Read C(i,j) into fast memory}
      {Read column j of B into fast memory}
      for k=1 to n
          C(i,j)=C(i,j) + A(i,k)*B(k,j)
      end for
      {Write C(i,j) back to slow memory}
   end for
end for
```

Unblocked Matrix Multiplication

```
for i=1 to n
    {Read row i of A into fast memory}
    for j=1 to n
        {Read C(i,j) into fast memory}
        {Read column j of B into fast memory}
        for k=1 to n
            C(i,j)=C(i,j) + A(i,k)*B(k,j)
        end for
        {Write C(i,j) back to slow memory}
    end for
end for
```

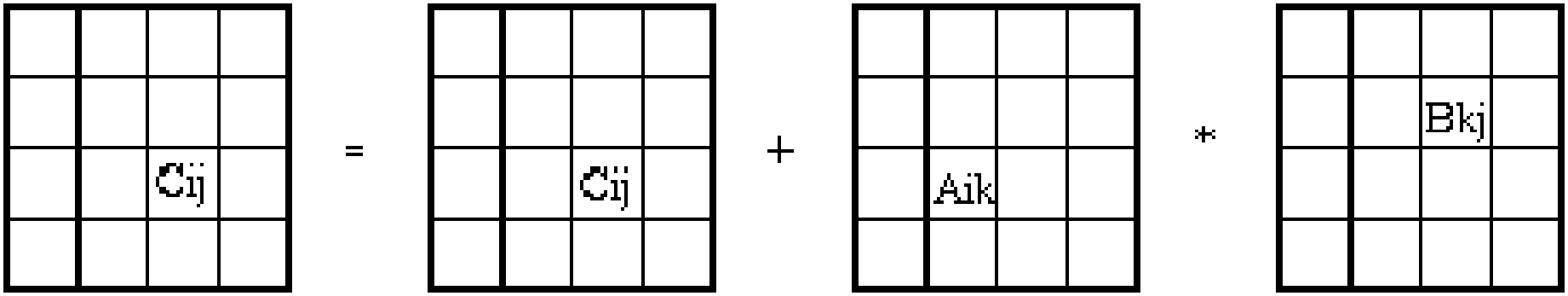
Let's count the number of accesses to the slow memory. For A, B and C respectively: $n^2, n^3, 2n^2$. Leading to:

$$m = n^3 + n^2 + 2n^2 = n^3 + 3n^2$$

and

$$q = \frac{f}{m} = \frac{2n^3}{n^3 + 3n^2} \sim 2.$$

Blocked Matrix Multiplication



$$C_{ij} = C_{ij} + \sum_{k=1}^{n} A_{ik} * B_{kj}$$

Blocked Matrix Multiplication

```
for i=1 to N
    for j=1 to N
        {Read Cij into fast memory}
        for k=1 to N
            {Read Aik into fast memory}
            {Read Bkj into fast memory}
            Cij = Cij + Aik * Bkj
        end for
        {Write Cij back to slow memory}
    end for
end for
```

Unblocked Matrix Multiplication

```
for i=1 to N
    for j=1 to N
        {Read Cij into fast memory}
        for k=1 to N
            {Read Aik into fast memory}
            {Read Bkj into fast memory}
            Cij = Cij + Aik * Bkj
        end for
        {Write Cij back to slow memory}
    end for
end for
```

Each matrix is regarded as N-by-N blocks of size $n/N \times n/N$.

For A, B and C we have respectively: $Nn^2, Nn^2, 2n^2$ accesses.

$$m = Nn^2 + Nn^2 + 2n^2 = (2N+2)n^2.$$

and

$$q = \frac{f}{m} = \frac{2n^3}{(2N+2)n^2} \sim \frac{n}{N} \sim \sqrt{\frac{M}{3}}.$$

Since 3 blocks must be in fast memory, so $N \sim n\sqrt{\frac{3}{M}}$.

Cilk

• Cilk has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group.

• Besides being used for research and teaching, Cilk was the system used to code the three world-class chess programs Tech, Socrates, and Cilkchess.

• Over the years, implementations of Cilk have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.

• Cilk is currently maintained by Matteo Frigo <athena@fftw.org>.

`http://supertech.csail.mit.edu/cilk/`

`http://www.cilk.com/`

Multithreaded Programming in Cilk

- Cilk software, user manual and relevant papers and events are available at http://supertech.csail.mit.edu/cilk/ and http://www.cilk.com/

- **Example program: Fibonacci**

C

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

Cilk

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

Multithreaded Programming in Cilk

Basic Cilk key words

```
cilk int fib(int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n − 1);
    y = spawn fib(n − 2);
    sync;
    return x + y;
  }
}
```

cilk:
Identifies a function as a Cilk procedure, capable of being spawned in parallel.

spawn:
The child procedure can be executed in parallel with the parent.

sync:
Cannot be passed until all spawned children have returned.

Exposing Parallelism in Cilk

• Cilk is a faithful extension of C, because the C elision of a Cilk program is a correct implementation of the semantics of the program.

• Cilk is a multithreaded language for parallel programming that generalizes the semantics of C by introducing linguistic constructs for parallel control.

• Moreover, on one processor, a parallel Cilk program scales down to run nearly as fast as its C elision.

Exposing Parallelism in Cilk

• The keyword `cilk` identifies a Cilk procedure, which is the parallel version of a C function:

- Parallelism is created when the invocation of a Cilk procedure is immediately preceded by the keyword `spawn`.
- Cilk's scheduler takes the responsibility of scheduling the spawned procedures on the processors of the parallel computer.

• A Cilk procedure cannot safely use the returned values of the children it has spawned until it executes a `sync` statement:

- When all its children return, execution of the procedure resumes at the point immediately following the `sync` statement.
- As an aid to programmers, Cilk inserts an implicit `sync` before every `return`.

The Cilk Terminology

• In Cilk terminology, a thread is a maximal sequence of instructions that ends with a `spawn`, `sync`, or `return` (either explicit or implicit) statement.

• The first thread that executes when a procedure is called is the procedure's initial thread, and the subsequent threads are successor threads.

• At runtime, the binary *spawn* relation causes procedure instances to be structured as a rooted tree, and the dependencies among their threads form a dag embedded in this spawn tree.

• The span is another name for the critical path.

Dynamic Multithreading in Cilk

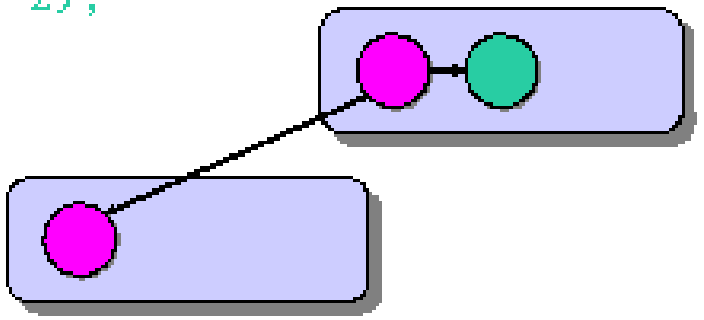
```
cilk int fib(int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n - 1);
```

Example: `fib(4)`

Dynamic Multithreading in Cilk

```
cilk int fib(int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n - 1);
    y = spawn fib(n - 2);
```

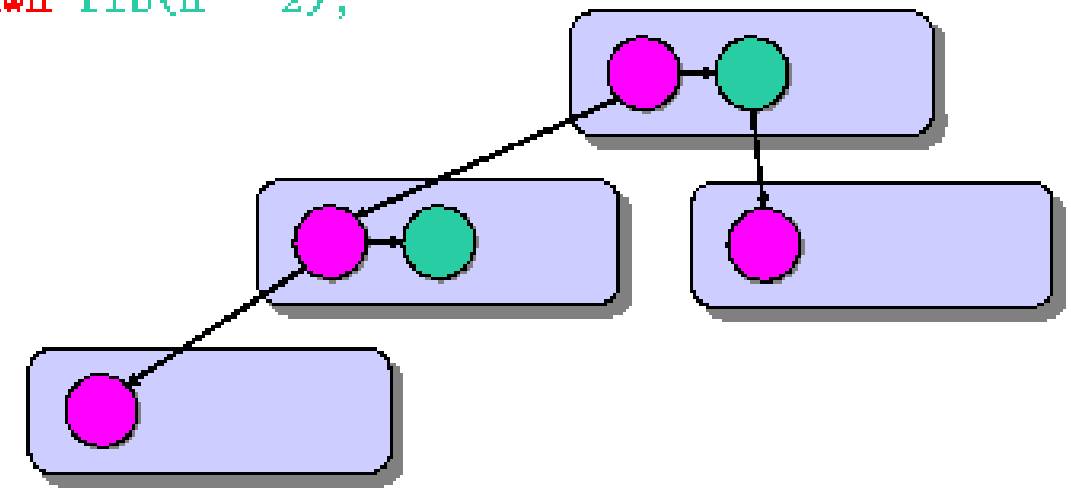
Example: `fib(4)`



Dynamic Multithreading in Cilk

```
cilk int fib(int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n - 1);
    y = spawn fib(n - 2);
    sync;
```

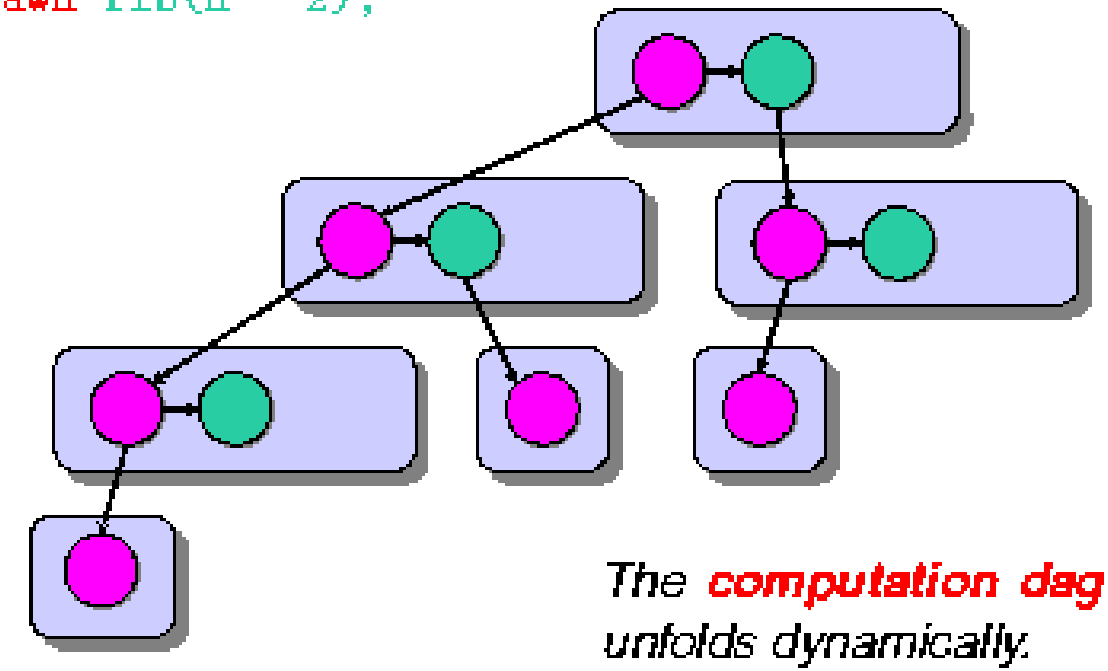
Example: `fib(4)`



Dynamic Multithreading in Cilk

```
cilk int fib(int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n - 1);
    y = spawn fib(n - 2);
    sync;
```

Example: `fib(4)`

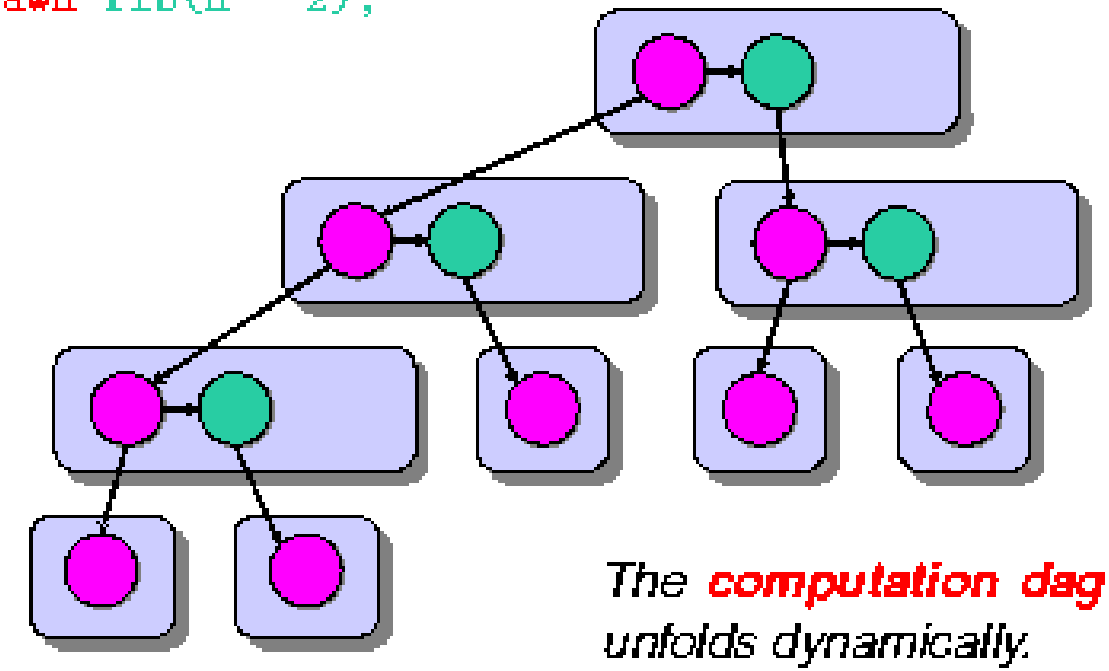


*The **computation dag** unfolds dynamically.*

Dynamic Multithreading in Cilk

```
cilk int fib(int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n - 1);
    y = spawn fib(n - 2);
    sync;
```

Example: fib(4)

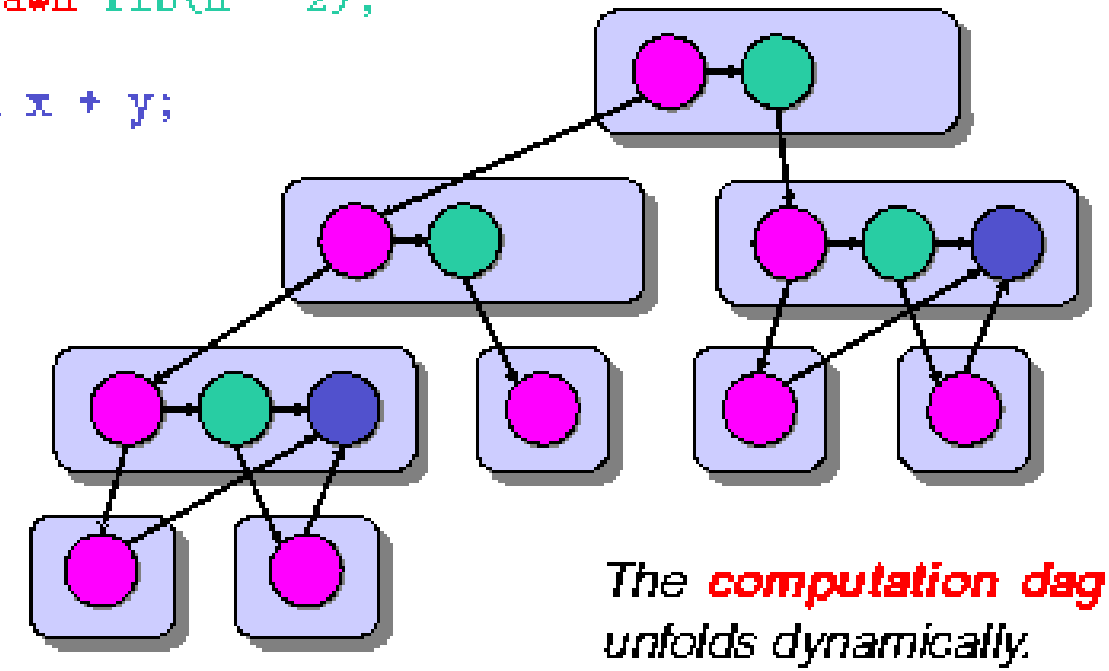


*The **computation dag** unfolds dynamically.*

Dynamic Multithreading in Cilk

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

Example: fib(4)

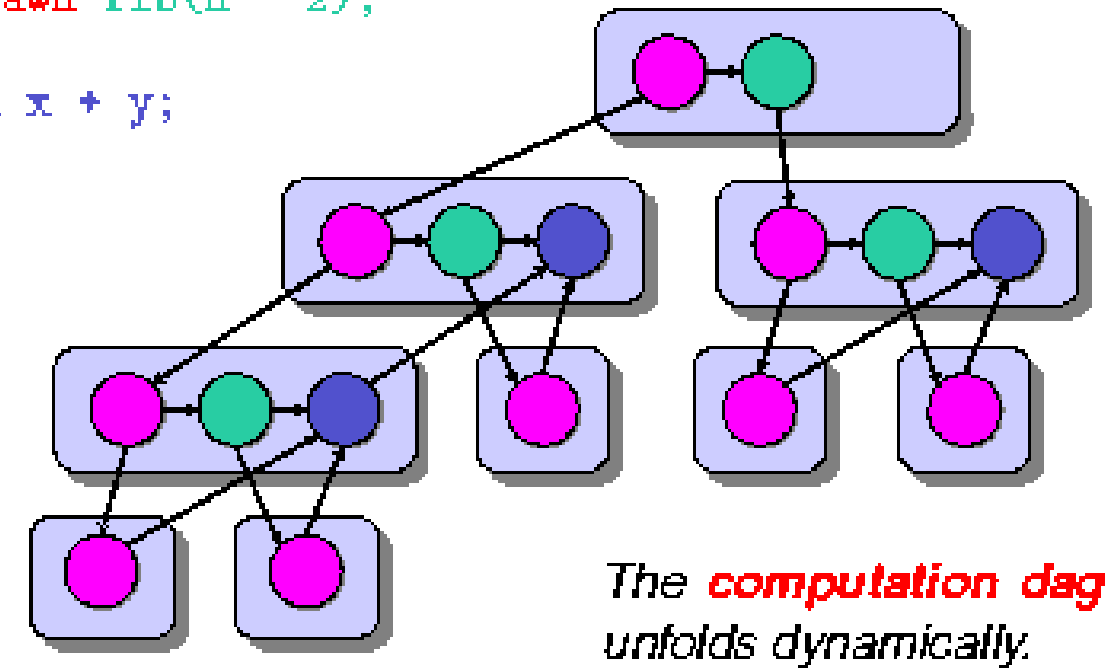


*The **computation dag** unfolds dynamically.*

Dynamic Multithreading in Cilk

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

Example: `fib(4)`

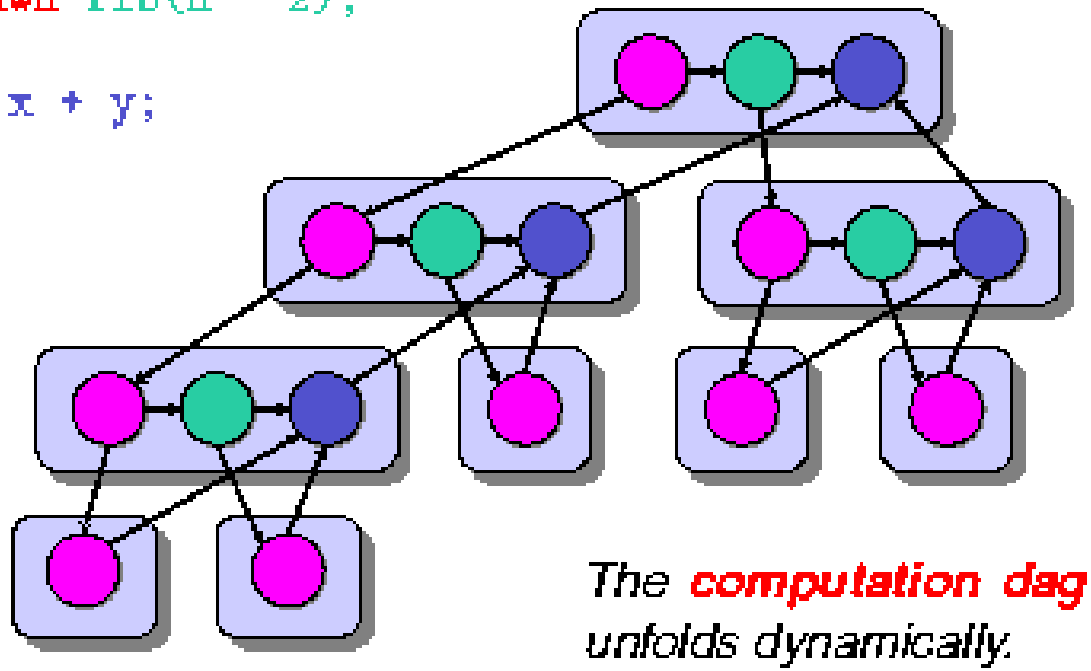


*The **computation dag** unfolds dynamically.*

Dynamic Multithreading in Cilk

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

Example: fib(4)



"Processor oblivious"

*The **computation dag** unfolds dynamically.*

Analysis of Multithreaded Algorithms

Definition 1 *We define several performance measures to evaluate various scheduler algorithms. We assume an ideal situation: no cache issues, no interprocessor costs:*

T_p *is the minimum running time on* p *processors.*

T_1 *is called the* **work**.

T_∞ *is the minimum running time with infinitely many processors.*

Observe that, on the previous slide, we have $T_1 = 17$ and $T_\infty = 8$.

Proposition 1 *Assuming that all threads run in unit time, the longest path in the DAG is equal to* T_∞. *For this reason,* T_∞ *is referred to as the* critical path length.

When threads do not run in unit time, then the vertices of the DAG need to be weighted by their individual runtimes.

Analysis of Multithreaded Algorithms

Proposition 2 *We have:* $T_p \geq T_1/p$.

Indeed, in the best case, p processors can do p works per unit of time.

Proposition 3 *We have:* $T_p \geq T_\infty$.

Indeed, $T_p < T_\infty$ contradicts the definitions of T_p and T_∞.

Definition 2 *A program's parallel execution plan can have:*

- linear speedup*:* $T_1/T_P = \Theta(P)$
- superlinear speedup*:* $T_1/T_P = \omega(P)$ *(not possible in this model, though it is possible in others).*
- sublinear speedup*:* $T_1/T_P = o(p)$

The Greedy Scheduler

Definition 3 *A* scheduler*'s job is to map a computation to particular processors:*

- *If decisions are made at runtime, the scheduler is* online*, otherwise, it is* offline*.*
- *A scheduler is* greedy *if attempts to do as much work as possible at every step.*

In any *greedy schedule*, there are two types of steps:

- **complete step**: There are at least p threads that are ready to run. The greedy scheduler selects any p of them and runs them.
- **incomplete step**: There are strictly less than p threads that are ready to run. The greedy scheduler runs them all.

The Greedy Scheduler

Theorem 1 (Graham, Brent) *Given p processors, a greedy scheduler executes any computation G with work T_1 and critical path length T_∞ in time:*

$$T_p \quad \leq \quad T_1/p \; + \; T_\infty .$$

Proof. Observe that

- There can be no more than T_1/p complete steps, by definition of T_1.
- There can be no more than T_∞ incomplete steps. Indeed:
  - (i) Let G' be the subgraph of G that remains to be executed immediately prior to a given incomplete step.
  - (ii) During this incomplete step, all threads that can be run are actually run
  - (iii) Hence removing this incomplete step from G' would reduce its the critical-path length by one.

□

The Greedy Scheduler

Corollary 1 *A greedy scheduler is always within a factor of 2 of optimal.*

PROOF. From Propositions 1 and 2, we have:

$$T_P \geq \max(T_1/P, T_\infty) \tag{1}$$

In addition, we can trivially express:

$$T_1/P \leq \max(T_1/P, T_\infty) \tag{2}$$

$$T_\infty \leq \max(T_1/P, T_\infty) \tag{3}$$

Given Theorem 1, we deduce:

$$\begin{aligned} T_P &\leq T_1/P + T_\infty && (4)\\ &\leq \max(T_1/P, T_\infty) + \max(T_1/P, T_\infty) && (5)\\ &\leq 2\max(T_1/P, T_\infty) && (6) \end{aligned}$$

which concludes the proof. □

Every step either brings to progress on the work (complete step), or on the critical path (incomplete step). A greedy scheduler is generally good.

The Greedy Scheduler

Corollary 2 *The greedy scheduler achieves linear speedup when* $T_\infty = O(T_1/p)$.

Proof.

$$\begin{aligned} T_p &\leq T_1/p + T_\infty && (7)\\ &= T_1/p + O(T_1/p) && (8)\\ &= \Theta(T_1/p) && (9) \end{aligned}$$

The idea is to operate in the range where T_1/P dominates T_∞. As long as T_1/P dominates T_∞, all processors can be used efficiently.

Pseudocode for Matrix Addition

- To multiply two $n \times n$ matrices A and B in parallel to produce a matrix C:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

- A procedure ADD to add $n \times n$ matrices (not in-place):

```
ADD(C, T, n)
1  if n = 1
2     then C[1,1] ← C[1,1] + T[1,1]
3     else  partition C and T into (n/2) × (n/2) submatrices
4           spawn ADD(C11, T11, n/2)
5           spawn ADD(C12, T12, n/2)
6           spawn ADD(C21, T21, n/2)
7           spawn ADD(C22, T22, n/2)
8           sync
```

Pseudocode for Matrix Multiplication

```
MULT(C, A, B, n)
 1  if n = 1
 2     then C[1,1] ← A[1,1] · B[1,1]
 3     else  allocate a temporary matrix T[1..n, 1..n]
 4           partition A, B, C, and T into (n/2) × (n/2) submatrices
 5           spawn MULT(C11, A11, B11, n/2)
 6           spawn MULT(C12, A11, B12, n/2)
 7           spawn MULT(C21, A21, B11, n/2)
 8           spawn MULT(C22, A21, B12, n/2)
 9           spawn MULT(T11, A12, B21, n/2)
10           spawn MULT(T12, A12, B22, n/2)
11           spawn MULT(T21, A22, B21, n/2)
12           spawn MULT(T22, A22, B22, n/2)
13           sync
14           spawn ADD(C, T, n)
15           sync
```

Performance of MULT

- Let $A_p(n)$ and $M_p(n)$ be the *P-processor* running time of ADD and MULT on $n \times n$ matrices respectively.
- The work for ADD can be expressed by the recurrence

$$A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2)$$

- The critical-path length for ADD is

$$A_\infty(n) = A_\infty(n/2) + \Theta(1) = \Theta(\lg n)$$

- The work for MULT is

$$M_1(n) = 8M_1(n/2) + A_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3)$$

- The critical-path length for MULT is

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$$

- The parallelism for MULT is

$$M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$$

Pseudocode for Matrix MULT-ADD

The MULT-ADD performs $C \leftarrow C + A$.

```
MULT-ADD(C, A, B, n)
 1  if n = 1
 2     then C[1,1] ← C[1,1] + A[1,1] · B[1,1]
 3     else  partition A, B, and C into (n/2) × (n/2) submatrices
 4           spawn MULT-ADD(C11, A11, B11, n/2)
 5           spawn MULT-ADD(C12, A11, B12, n/2)
 6           spawn MULT-ADD(C21, A21, B11, n/2)
 7           spawn MULT-ADD(C22, A21, B12, n/2)
 8           sync
 9           spawn MULT-ADD(C11, A12, B21, n/2)
10           spawn MULT-ADD(C12, A12, B22, n/2)
11           spawn MULT-ADD(C21, A22, B21, n/2)
12           spawn MULT-ADD(C22, A22, B22, n/2)
13           sync
```

Performance of MULT-ADD

- Let $MA_p(n)$ be the *P-processor* running time of MULT-ADD.
- The work for MULT-ADD is

$$MA_1(n) = \Theta(n^3)$$

- The critical-path is

$$MA_\infty(n) = 2MA_\infty(n/2) + \Theta(1) = \Theta(n)$$

- The parallelism for MULT-ADD is

$$MA_1(n)/MA_\infty(n) = \Theta(n^2)$$

- Besides, saving space often saves time due to hierarchical memory.

Pseudocode for Merge Sort

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          spawn MERGE-SORT(A, p, q)
4          spawn MERGE-SORT(A, q + 1, r)
5          sync
6          MERGE(A, p, q, r)
```

- The work of MERGE-SORT on an array of n elements is

$$T_1(n) = 2T_1(n/2) + \Theta(n) = \Theta(n \lg n)$$

- The running time of MERGE is $\Theta(n)$.
- The critical-path length of MERGE-SORT is

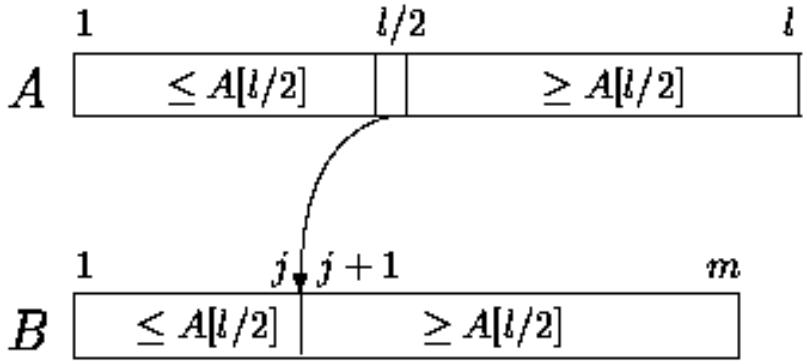
$$T_\infty(n) = T_\infty(n/2) + \Theta(n) = \Theta(n)$$

- The parallelism of MERGE-SORT is

$$T_1(n)/T_\infty(n) = \Theta(\lg n)$$

- The obvious bottleneck is MERGE.

A Parallel Merge: P-MERGE




```
P-MERGE(A[1..l], B[1..m], C[1..n])
 1  if m > l      ▷ without loss of generality, larger array should be first
 2     then spawn P-MERGE(B[1..m], A[1..l], C[1..n])
 3  elseif n = 1
 4     then C[1] ← A[1]
 5  elseif l = 1      ▷ and m = 1
 6     then if A[1] ≤ B[1]
 7             then C[1] ← A[1]; C[2] ← B[1]
 8             else C[1] ← B[1]; C[2] ← A[1]
 9     else find j such that B[j] ≤ A[l/2] ≤ B[j + 1] using binary search
10          spawn P-MERGE(A[1..(l/2)], B[1..j], C[1..(l/2 + j)])
11          spawn P-MERGE(A[(l/2 + 1)..l], B[(j + 1)..m], C[(l/2 + j + 1)..n])
12  sync
```

Performance of P-Merge

- Let $PM_p(n)$ be the *P-processor* running time of P-Merge.
- In the worst case, the critical-path length of P-Merge is

$$PM_\infty(n) = PM_\infty(3n/4) + \Theta(\lg n) = \Theta(\lg^2 n)$$

- The worst-case work of P-Merge satisfies the recurrence

$$PM_1(n) = PM_1(\alpha n) + PM_1((1-\alpha)n) + \Theta(\lg n)$$

  , where α is a constant in the range $1/4 \le \alpha \le 3/4$.

- We assume inductively that $PM_1(n) \le an - b\lg n$ for some constants $a, b > 0$.

$$\begin{aligned} PM_1(n) &\le a\alpha n - b\lg(\alpha a) + a(1-\alpha)n - b\lg((1-\alpha)n) + \Theta(\lg n) \\ &= an - b(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\ &= an - b(\lg\alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\ &= an - b\lg n - (b(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\ &\le an - b\lg n \end{aligned}$$

Performance of P-Merge

- We can pick a large enough to satisfy the base conditions and choose b large enough so that $b(\lg n + \lg(\alpha(1-\alpha))$ dominates $\Theta(\lg n)$, and thus $PM_1(n) = \Theta(n)$.
- The worst case critical-path length of the Merge-Sort now satisfies

$$T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n) = \Theta(\lg^3 n)$$

- The parallelism is now $\Theta(n \lg n)/\Theta(\lg^3 n) = \Theta(n \lg^2 n)$.

Installing Cilk

```
tar xvfz cilk-5.4.6.tar.gz

./configure

make

make install

make distclean
```

This will install

- libraries in `/usr/local/lib`
- header files in `/usr/local/include`
- compiler in `/usr/local/bin`

Compiling and Running Cilk Programs

Basic compilation command line:

```
cilkc -O2 fib.cilk -o fib
```

Basic running command line:

```
fib --nproc 2 30
```

Collecting profiling information (processor activity, thread migration, memory allocation) and computing the span:

```
cilkc -cilk-profile -cilk-span -O2 fib.cilk -o fib
fib --nproc 4 --stats 1 30
```

This should yield:

```
Result: 832040
RUNTIME SYSTEM STATISTICS:
Wall-clock running time on 4 processors: 2.5932
Total work = 10.341069 s
Total work (accumulated) = 7.746886 s
Span = 779.588000 us    Parallelism = 9937.154003
```

Observing Speed-up

```
./strassen --nproc 1 -n 256

Cilk Example: strassen
              running on 1 processor

Running time  = 0.044962 s

./strassen --nproc 2 -n 256

Cilk Example: strassen
              running on 2 processors

Running time  = 0.025789 s
```

Computing Critical Path and Work

```
cilkc -cilk-profile -cilk-span -O2 -I . getoptions.o strassen.cilk -o s

./strassen-profile --nproc 2 -stats 1 -n 256

Running time  = 0.029277 s
Work          = 0.047263 s
Critical path = 0.005524 s

/strassen-profile --nproc 1 -stats 1 -n 256

Running time  = 0.046969 s
Work          = 0.046550 s
Critical path = 0.005556 s
```

Storage Allocation

• Like the C language on which it is based, Cilk provides two types of memory: stack and blue heap.

• Stack memory is allocated by the run-time system for procedure-local variables and arrays:

• Stack memory is automatically deallocated

- when the Cilk procedure or,
- when a C function

that allocated the memory returns.

• Heap memory is allocated by the usual `malloc()` library function and deallocated with `free()`.

Storage Allocation

- Cilk supports C's rule for pointers:
  - a pointer to stack space can be passed from parent to child, but not from child to parent.
  - Pointers can be passed upward only if they reference data stored on the heap (allocated with *malloc*).
- Functions cannot return references to local variables.
- When procedures are run in parallel by Cilk, the running threads operate on their view of the call stack.

Cilk's Cactus Stack

• A **cactus stack** is used to implement C's rule for sharing of function-local variables.

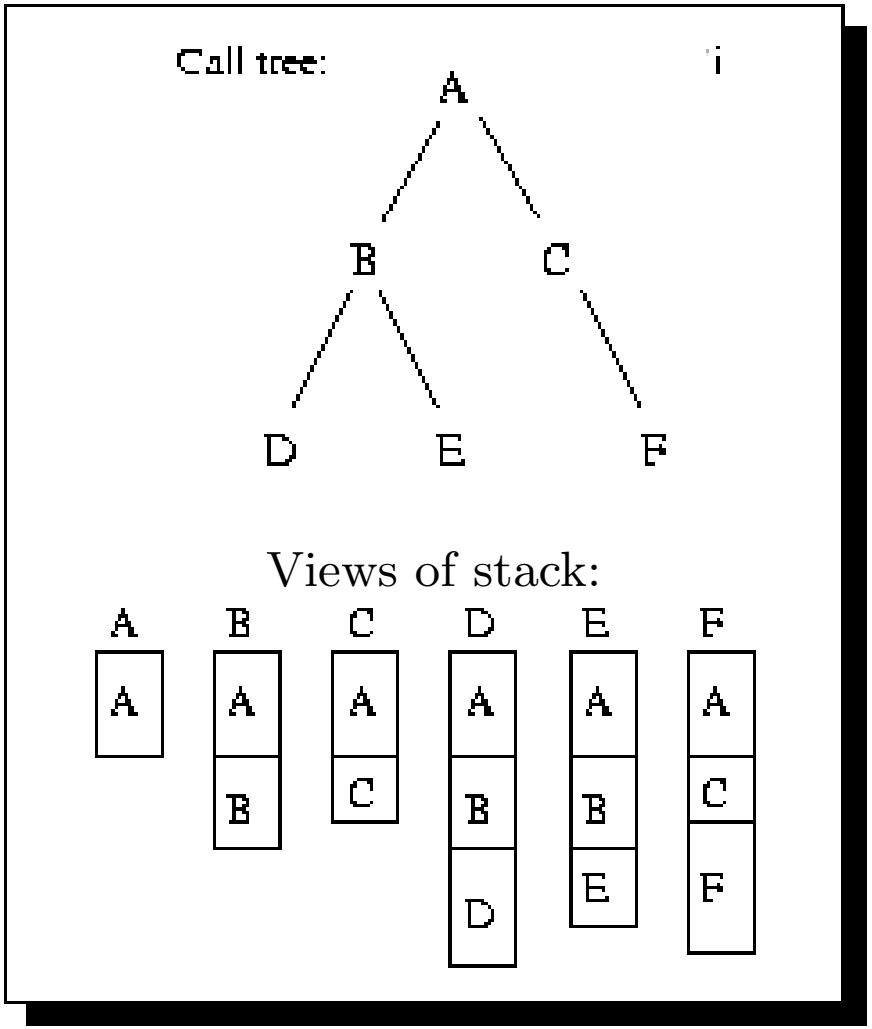
• A stack frame can only see data stored in the current and in the previous stack frames.

```
void A(void)
{ B();
  C(); }

void B(void)
{ D();
  E(); }

void C(void)
{ F(); }

void D(void) {}
void E(void) {}
void F(void) {}
```



Shared Memory

- Sharing occurs
  - when a global variable is accessed by procedures operating in parallel,
  - from the passing of pointers to spawned procedures, allowing more than one procedure to access the object addressed by the pointer.

- The updating of shared objects in parallel is called race condition. This can cause nondeterministic anomalies!

- The easiest way to deal with the anomalies of shared access is simply to avoid writing code in which one thread modifies a value that might be read or written in parallel by another thread.

- Cilk provides different mechanisms to detect or avoid race conditions.

Race Condition

- This is the most prominent obstacle for parallel programming
- An example in Cilk:

```
cilk int foo(void)
{
    int x = 0;
    spawn bar(&x);
    spawn bar(&x);
    sync;
    return x;
}
```

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

If this were a serial code, we would expect that foo returns 2.

In assembly:

```
read x
add
write x
```

Race Condition

In parallel execution:

```
bar 1:
read x (1)
add
write x (2)

bar 2:
read x (3)
add
write x (4)
```

If it is executed in the order of (1) (2) (3) (4), then foo returns 2;
If it is executed in the order of (1) (3) (2) (4), then foo would return 1.
A race condition!

Locking

• Cilk provides mutual exclusion locks that allow you to create atomic regions of code. A lock has type `Cilk_lockvar`.

• The two operations on locks are:

- `Cilk_lock` to test a lock and block if not already acquired, and
- `Cilk_unlock` to release a lock.

Both functions take a single argument which is an object of type `Cilk_lockvar`.

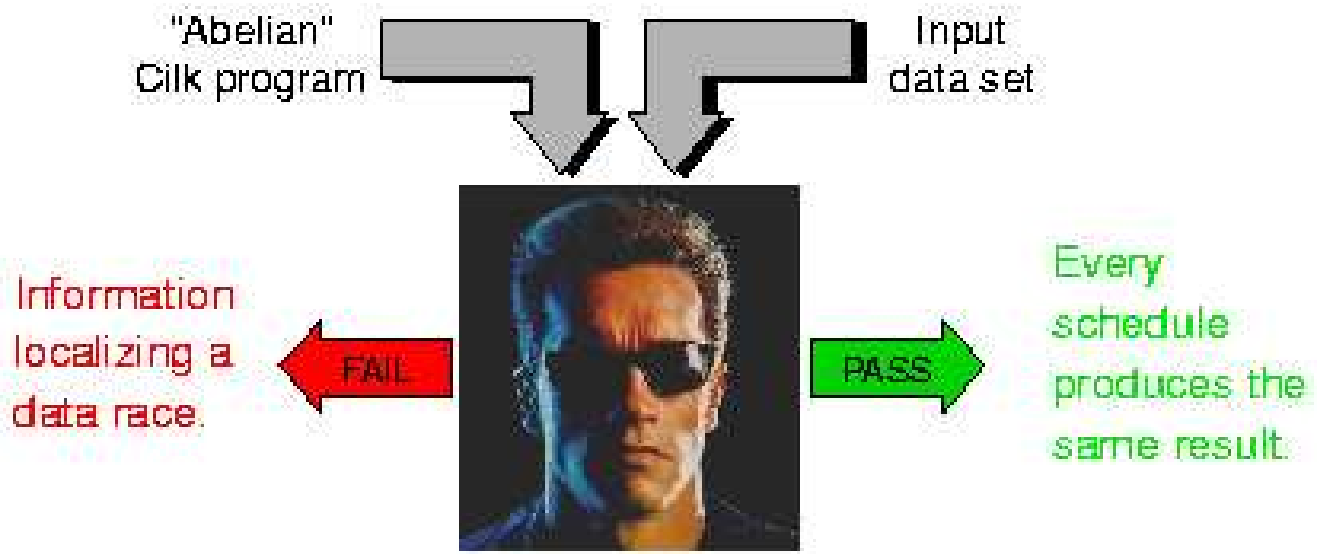
• The lock object must be initialized using `Cilk_lock_init()` before it is used.

• The region of code between a `Cilk_lock` statement and the corresponding `Cilk_unlock` statement is called a *critical section.*

• A critical section of code accessing shared data is protected, so that other threads which read from or write to the chunk of data are excluded from running.

Detecting Race Condition

• **Nondeterminator**: a tool in Cilk to help check for race conditions. Cilk's Nondeterminator debugging tool provably guarantees to detect and localize data-race bugs.



• **Further detail** refers to "Nondeterminator-3: A Provably Good Data-Race Detector That Runs in Parallel", Masters' Thesis by Tushara C. Karunaratna.

Using an inlet

```
cilk int fib(int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n − 1);
    y = spawn fib(n − 2);
    sync;
    return (x + y);
  }
}
```

```
cilk int fib(int n) {
  int x = 0;
  inlet void summer (int result)
  {
    x += result;
    return;
  }
  if (n < 2) return n;
  else {
    summer(spawn fib(n − 1));
    summer(spawn fib(n − 2));
    sync;
    return (x);
  }
}
```

Using an inlet

- An inlet is essentially a C function internal to a Cilk procedure.

- In the normal syntax of Cilk, the spawning of a procedure must occur as a separate statement and not in an expression, **unless the spawn is performed as an argument to an inlet call**. In this case:
  - the procedure is spawned, and when it returns, the inlet is invoked.
  - In the meantime, control of the parent procedure proceeds to the statement following the inlet call.
  - No lock is required around the accesses to `x` by `summer`, because Cilk provides atomicity implicitly.
  - An **inlet** is precluded from containing `spawn` and `sync` statements, and thus it operates atomically as a single thread.

Using an abort

• Sometimes, a procedure spawns off parallel work which is later discovered to be unnecessary.

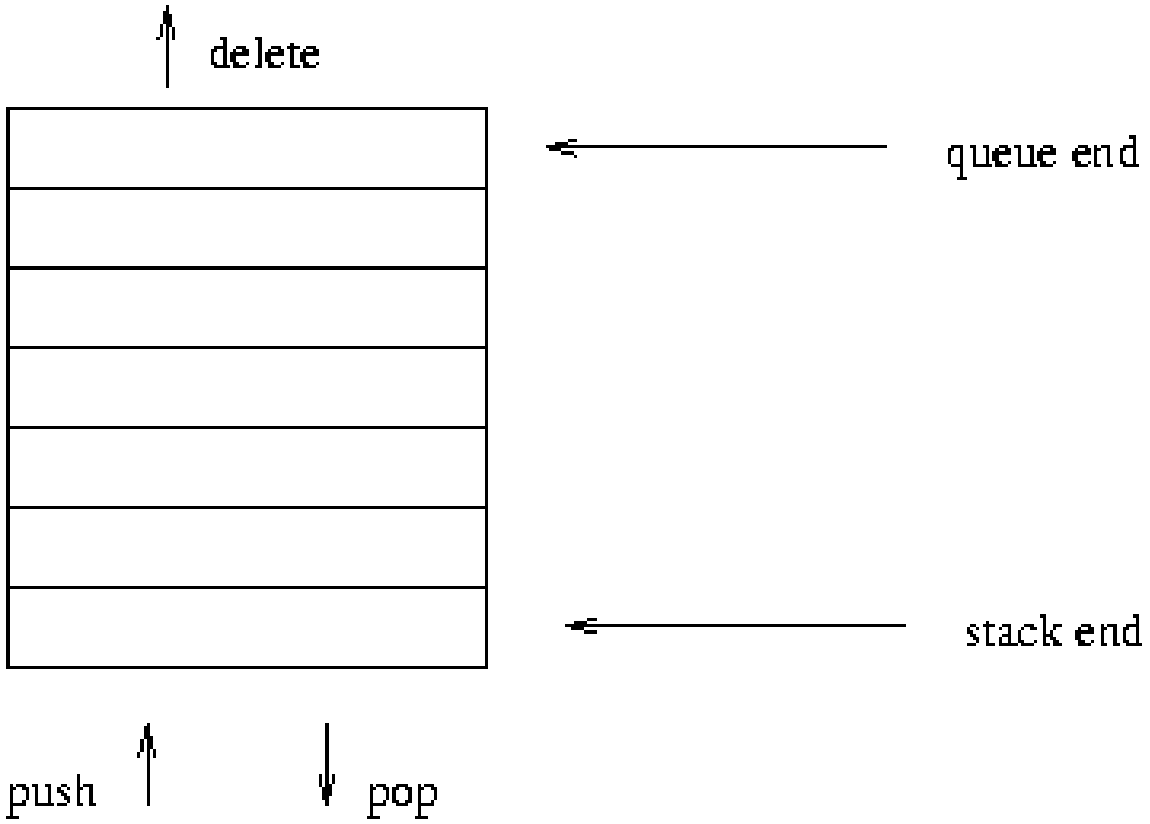
• This *speculative work* can be aborted in Cilk using the `abort` primitive inside an `inlet`.

• The `abort` statement, when executed inside an `inlet`, causes all of the already-spawned children of the procedure to terminate.

Cilk's Thread Scheduler

- Cilk's randomized work-stealing scheduler load-balances the computation at run-time.
- A mathematical proof guarantees near-perfect linear speed-up on applications with sufficient parallelism, as long as the architecture has sufficient memory bandwidth.
- A Cilk program running on 1 processor typically exhibits a negligible slowdown compared with its C elision.
- A `spawn/return` in Cilk is over 450 times faster than a Pthread `create/exit` and less than 3 times slower than an ordinary C function call on a modern Intel processor.

Cilk's Work-stealing Scheduler: the Ready Deque

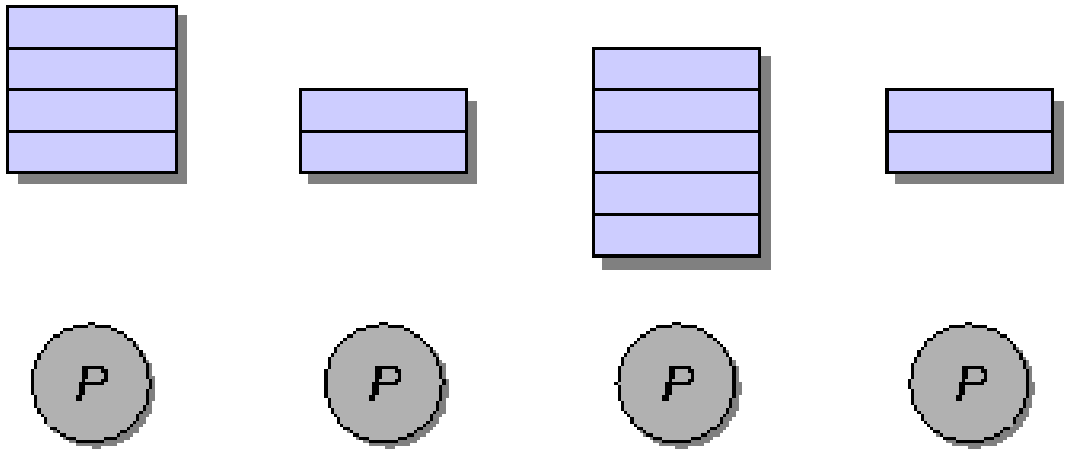


- Each processor maintains a data structure called a ready deque.
- A ready deque is a double ended queue.
- Every level of the ready deque contains a procedure instance that is ready to execute.

Cilk's Operations on the Deque

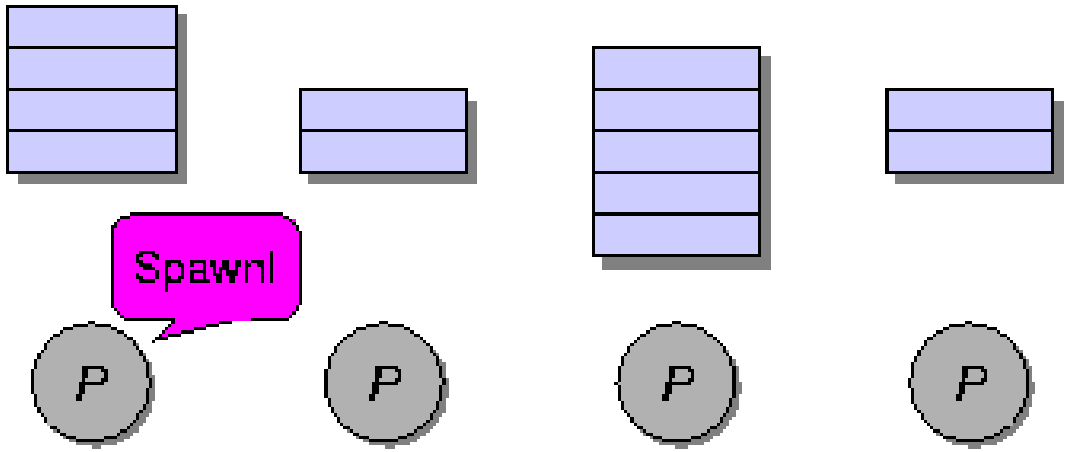
- Adding a procedure instance to the bottom of the deque represents a procedure call being spawned.
- A procedure instance being deleted from the bottom of the deque represents the processor beginning/resuming execution on that procedure.
- Deletion from the top of the deque corresponds to that procedure instance being stolen.
- Cilk begins executing the user program by initializing all ready deques to be empty, placing the root thread into the level-0 list of Processor 0's deque and then starting a scheduling loop on each processor.

Cilk's Work-stealing Scheduler



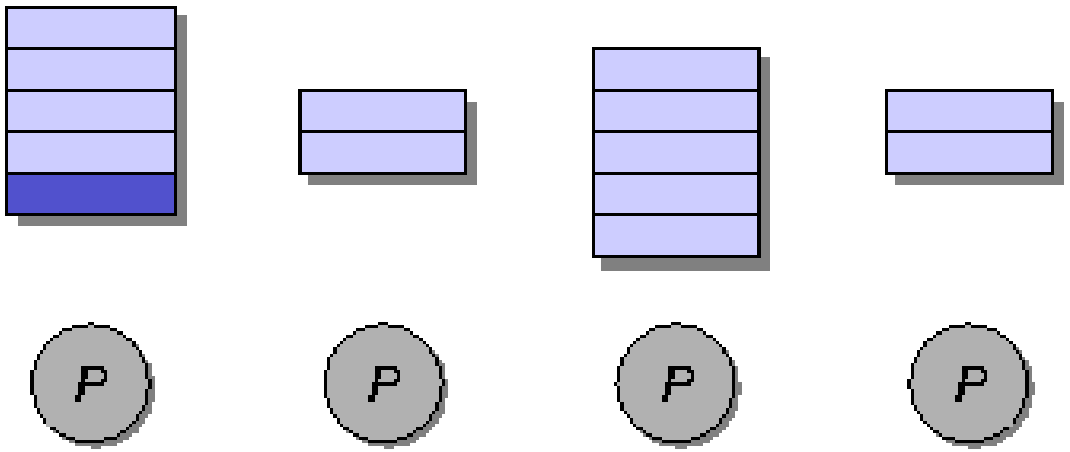
This picture illustrates some stage in the middle of a Cilk program execution on 4 processors. We number them from left to right as Processor 1, 2, 3 and 4. Each processor has a ready deque with a number of procedure instances being ready to be executed.

Cilk's Work-stealing Scheduler



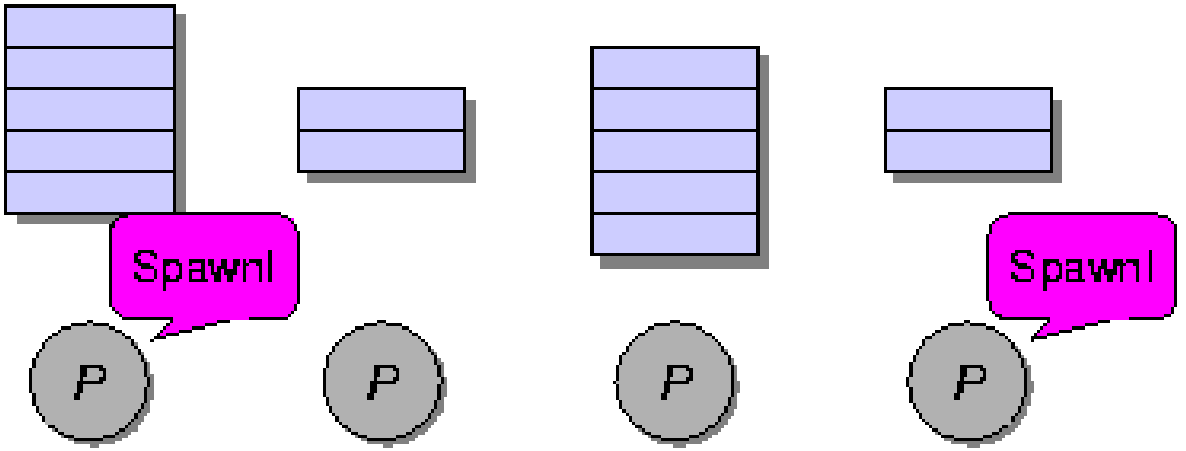
Processor 1 spawns a new procedure instance.

Cilk's Work-stealing Scheduler



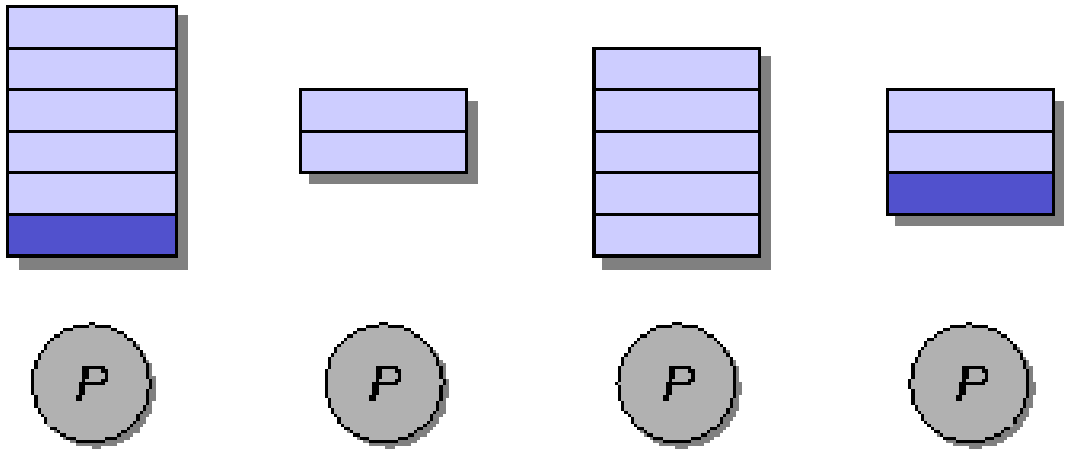
Processor 1 pushes this procedure instance to the bottom of its ready deque.

Cilk's Work-stealing Scheduler



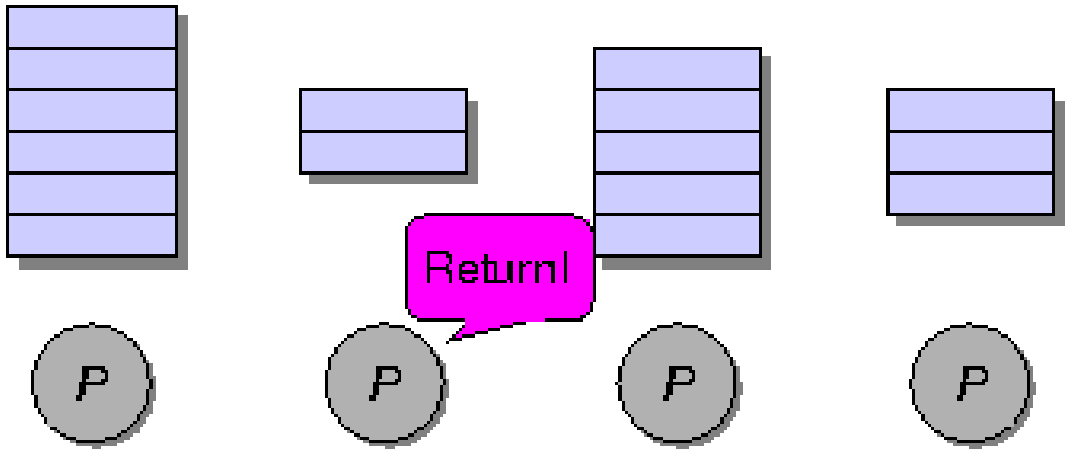
Both Processor 1 and Processor 4 are spawning.

Cilk's Work-stealing Scheduler



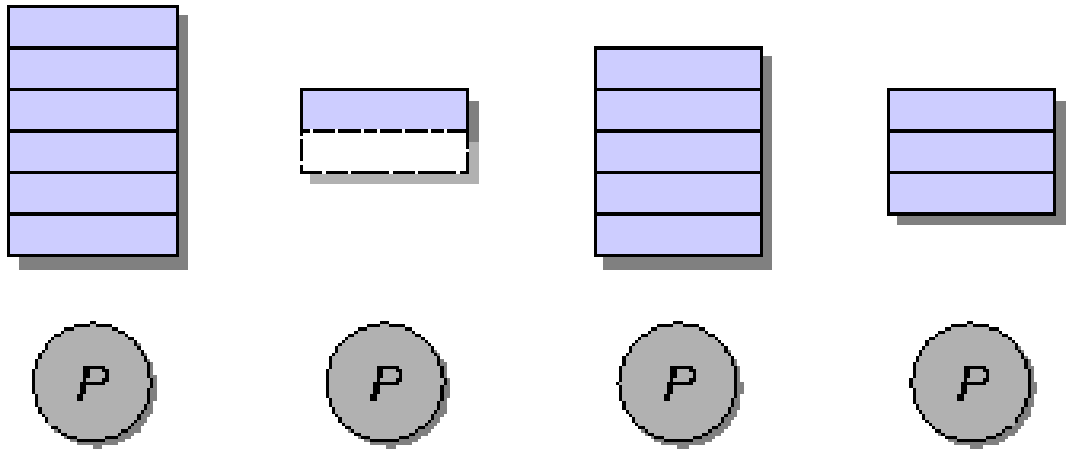
Each of Processor 1 and Processor 4 pushes its new procedure instance to the bottom of its deque.

Cilk's Work-stealing Scheduler



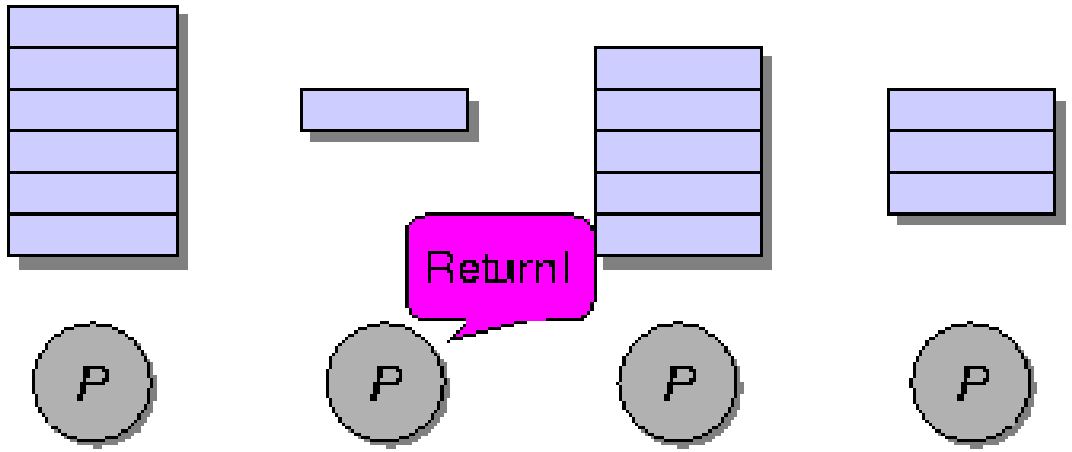
Processor 2 finishes executing a procedure instance and returns.

Cilk's Work-stealing Scheduler



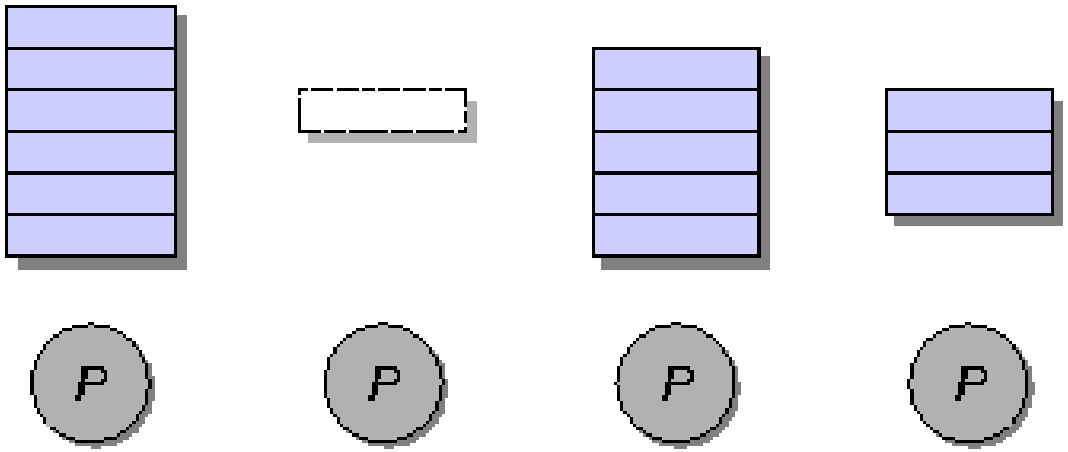
Processor 2 deletes this procedure instance from the bottom of its deque.

Cilk's Work-stealing Scheduler



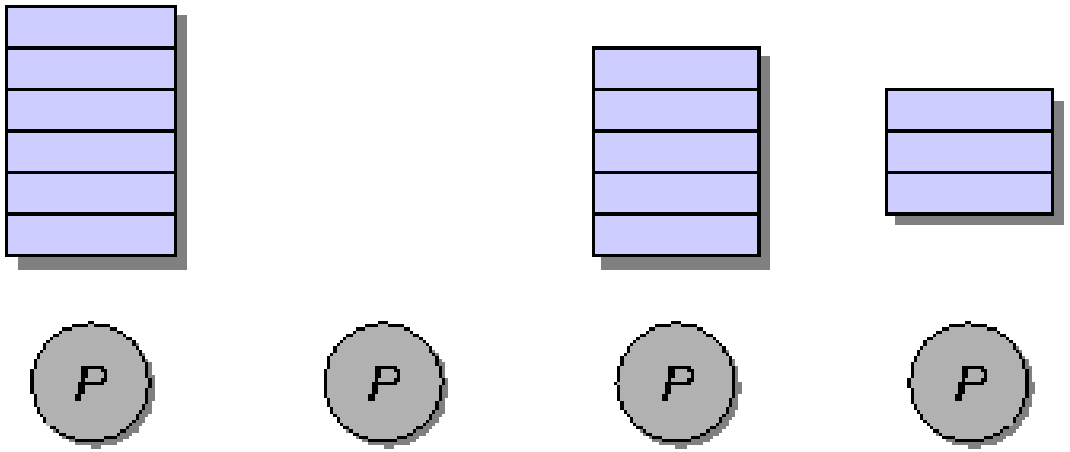
Processor 2 finishes executing another procedure instance and returns.

Cilk's Work-stealing Scheduler



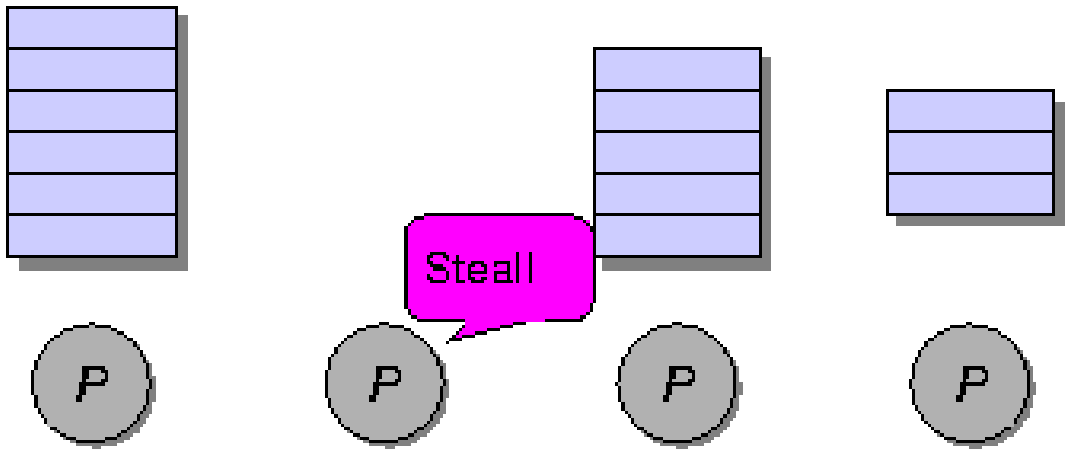
Processor 2 deletes this procedure instance from the bottom of its deque.

Cilk's Work-stealing Scheduler



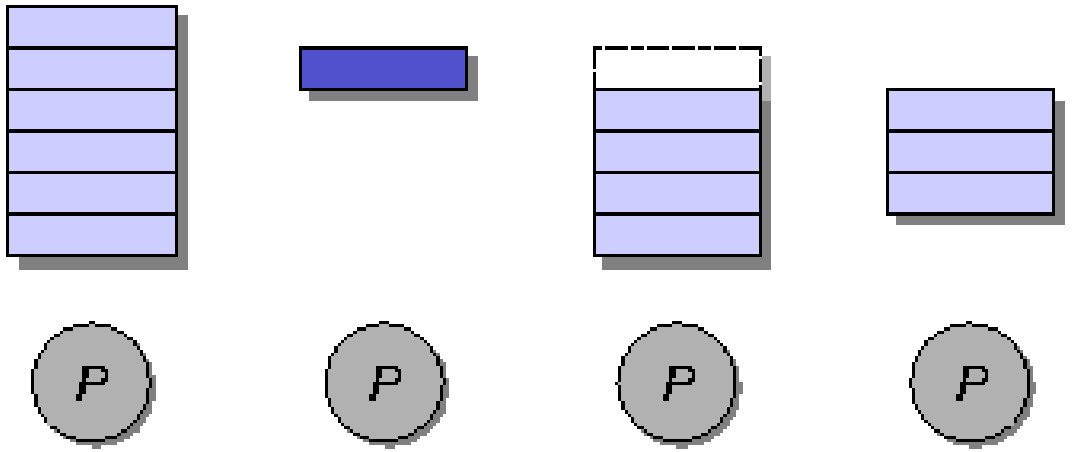
Processor 2's ready deque is empty. No job to do now!

Cilk's Work-stealing Scheduler



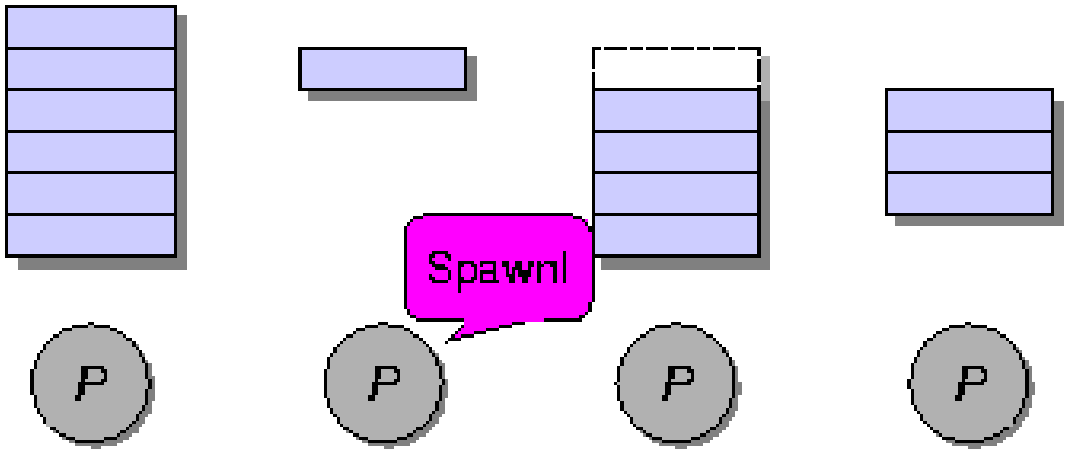
Processor 2 will steal one procedure instance from the other processors at random.

Cilk's Work-stealing Scheduler



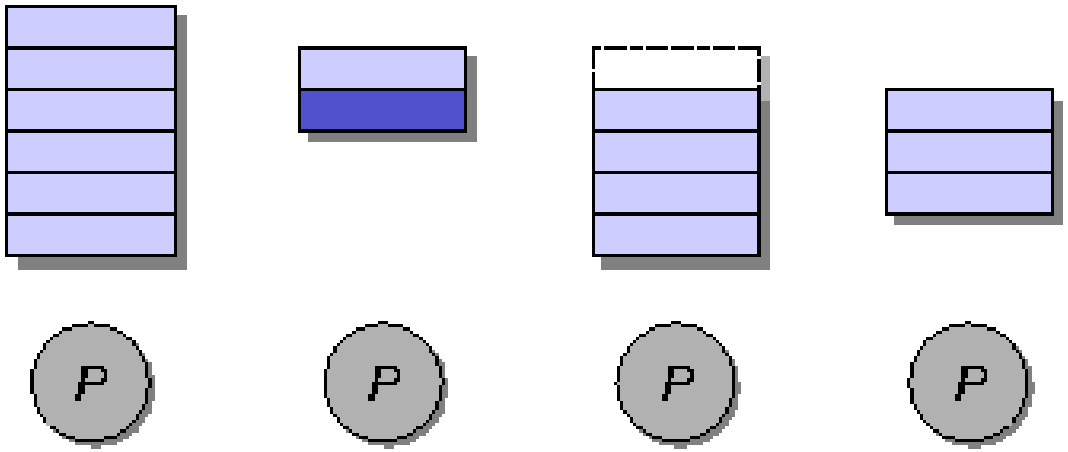
Processor 2 steals one procedure instance from the top of the deque of Processor 3.

Cilk's Work-stealing Scheduler



Processor 2 spawns a new procedure instance.

Cilk's Work-stealing Scheduler



Processor 2 pushes this procedure instance to the bottom of its deque.

The Work-stealing Algorithm

When a processor p begins work stealing (then it is called a **thief**), it operates as follows:

(1) Processor p chooses a victim processor uniformly at random from the set of all processors. Let the victim processor be v.

(2) If processor v's deque is empty, processor p repeats Step (1).

(3) Otherwise if processor v's deque is not empty, processor p steals the top procedure instance from v's deque. This is the reason the deque data structure must support pops from the top. Processor p then begins to work on this stolen procedure instance.

Performance of Work-stealing

Theorem 2 *On P processors, Cilk's work-stealing scheduler achieves an expected running time of*

$$T_P = T_1/P + O(T_\infty)$$

Proof sketch. Three underlying hypotheses:

(H_1) each Cilk thread executes in unit time,

(H_2) for almost all "parallel steps" there are (at least) P of threads to run.

(H_3) a processor is either working or stealing.

- The number of complete parallel steps (steps at which stealing is not needed since all processors have work on their deque) is at most T_1/P;
- the number of incomplete parallel steps is at most T_∞; at each incomplete step, a thief may reduce by 1 the running time with a probability of $1/P$; thus he expected number of (successful) steals is $O(PT_\infty)$.

Since there are P processors, the expected time is

$$(T_1 + O(PT_\infty))/P = T_1/P + O(T_\infty).$$

Critical Path Overhead

- The critical-path overhead is the smallest constant c_∞ such that

$$T_p \leq T_1/P + c_\infty T_\infty .$$

- The average parallelism is $\overline{P} = T_1/T_\infty$, which corresponds to the maximum possible speed-up that the application can obtain.

- The assumption of parallel slackness is that

$$\overline{P}/P >> c_\infty ,$$

that is, P is much smaller than the average parallelism .

- Under the assumption it follows that $T_1/P >> c_\infty T_\infty$, thus

c_∞ has little effect on performance when sufficiently slackness exists.

Work and Space Overheads

- Let T_s be the running time of the C elision of a Cilk program.
- Then, we denote by c_1 the work overhead

$$c_1 = T_1/T_s$$

- Recall the expected running time: $T_P \leq T_1/P + c_\infty T_\infty$. Thus we get

$$T_P \leq c_1 T_s/P + c_\infty T_\infty \simeq c_1 T_s/P.$$

- We can now restate the work first principle precisely:

Minimize c_1 , even at the expense of a larger c_∞.

Theorem 3 *The space S_p of a parallel execution on P processors required by Cilk's work-stealing satisfies:*

$$S_p \leq P \cdot S_1 \ , \tag{10}$$

where S_1 is the minimal serial space requirement.

Implementation of Work-stealing

• In Cilk's implementation, both victim and thief operate directly through shared memory on the victim's ready deque.

- This mechanism reduces work overhead (w.r.t. signaling interrupts)
- The crucial issue is how
  – to resolve the race condition that arises when a thief tries to steal the same frame that its victim is attempting to pop.
  – without enforcing the victim to use a lock before popping.

• **Hypotheses:** Only variable-reads and variable-writes are atomic.

• **Shared variables:**

- Three atomic shared variables `T`, `H`, and `E` are attached to each deque.
- To arbitrate among different thieves attempting to steal from the same victim, a hardware lock `L` is used:
  – This overhead can be amortized against the critical path.

Implementation of Work-stealing

Remarks

- Cilk adopted Dijkstra's protocol for mutual exclusion, which assumes only the reads and writes are atomic.
- The key idea is that actions by the worker on the tail of the queue contribute to work overhead, while actions by thieves on the head of the queue contribute to critical-path overhead.
- In accordance with the work-first principle, they attempt to move costs from the worker to the thief.
- We present below a simplified protocol, "TH" protocol, that uses only two shared variables `T` and `H` designating the tail and the head of the deque respectively.
- See the Cilk papers for the complete "THE" protocol.

The TH Protocol

```
1: push() {
2:   T++;
3: }
4: pop() {
5:    T--;
6:    if (H > T) {
7:        T++;
8:        lock(L);
9:        T--;
10:       if (H > T) {
11:           T++;
12:           unlock(L);
13:           return FAILURE;
14:       }
15:       unlock(L);
16:   }
17:   return SUCCESS;
18:}
```

(a)

```
1: steal() {
2:    lock(L);
3:    H++;
4:    if (H > T) {
5:        H--;
6:        unlock(L);
7:        return FAILURE;
8:    }
9:    unlock(L);
10:   return SUCCESS;
11: }
```

(b)

Figure 1: **(a)** Pseudocode of the actions performed by the worker/victim in the TH protocol. **(b)** Pseudocode of the actions performed by the thief in the TH protocol.

The TH Protocol

- The pseudocode assumes that the deque is implemented as an array of frames (= procedure instances together with their "environments").
- The index T (= tail = bottom) points to the first unused element in the array, and H (= head) points to the first frame on the deque.
- Indices grow from the head towards the tail, and most of the time, $\mathsf{T} \geq \mathsf{H}$.
- The worker pushes and pops frames by altering T, while the thief only increments H and does not alter T.
- The lock L ensures that only one thief can steal from the deque at a time.
- The push operation is always safe, because it does not involve any interaction between a thief and its victim.
- For the pop operation, there are three cases, shown in Figure 2 in the next slide.

The TH Protocol

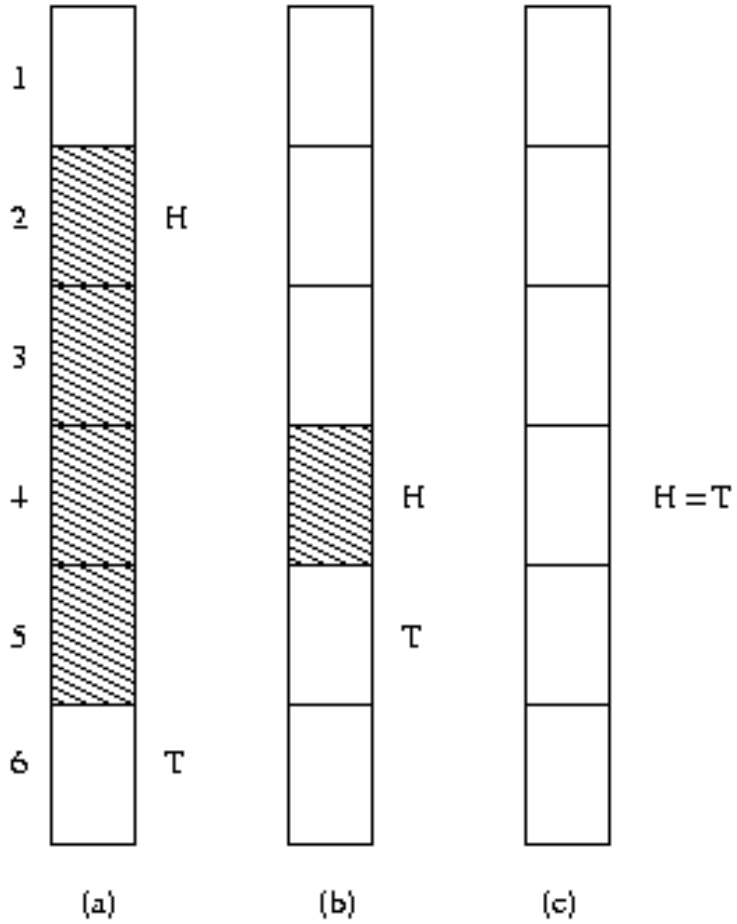


Figure 2: The three cases of the ready deque in the TH protocol. A shaded block indicates the presence of a frame at a certain position in the deque. The head and the tail are marked by T and H.

The THE Protocol

(a) Both the thief and the victim attempt to obtain different frames from the deque concurrently. In this case both are successful, and they do not interfere.

(b) The deque contains only one frame. Either victim and thief will get the frame if the other is not making an attempt to obtain it. If both victim and thief try to get the frame, the protocol guarantees that at least one of them discovers that $\mathsf{T} > \mathsf{H}$.

- If the thief discovers that $\mathsf{T} > \mathsf{H}$, it restores H to its original value and retreats.
- If the victim discovers $\mathsf{T} > \mathsf{H}$, it will restart the protocol after acquiring the lock L, which guarantees that it will get the frame without interference from any thief unless a thief has already stolen the frame.

(c) The deque is empty. A thief always fails to steal, and the victim also fails to pop the frame. The control passes to the Cilk runtime system.

The TH Protocol

- To summarize, the TH protocol contributes little to the work overhead.
- Pushing only involves updating T.
- Successfully popping a frame involves only 6 operations — 2 memory loads, 1 memory store, 1 decrement, 1 comparison, and 1 (predictable) conditional branch.
- In the case where both thief and victim simultaneously try to grab the same frame, the cost incurred by the lock can be considered as part of the critical-path overhead and does not influence the work overhead.

Matrix Multiplication!

Three implementations of classical matrix multiplication:

- Cilk example programs. http://supertech.csail.mit.edu/cilkImp.html
- `matmul.cilk`: uses two synchronization points but does not require intermediate allocation.
- `rectmul.cilk`: tries to avoid the second synchronization point but may require intermediate allocation; relies on a 2-recursive calls scheme instead of 8. Block size is 16. Use register hints in base computation.
- `spacemul.cilk`: tries to avoid the second synchronization point but may require intermediate allocation; relies on a classical 8-recursive calls approach. Block size is 16. Use register hints in base computation.
- **Which one is better? Why?**

Matrix Multiplication on 2 Processors

matmul/dim	128	256	512	1024
Running time	0.021	0.116	0.957	6.755
Work	0.028	0.207	1.699	11.650
Critical path	0.003	0.010	0.037	0.138
MFLOPS	197.518	288.506	280.378	317.905

rectmul/dim	128	256	512	1024
Running time	0.019	0.151	1.143	8.739
Work	0.027	0.218	1.721	12.785
Critical path	0.0004	0.001	0.0008	0.001
MFLOPS	214.318	221.679	234.528	245.591

spacemul/dim	128	256	512	1024
Running time	0.017	0.091	0.650	4.693
Work	0.019	0.132	0.992	6.789
Critical path	0.0007	0.001	0.001	0.002
MFLOPS	236.776	364.659	412.516	457.286

Matrix Multiplication on 4 Processors

matmul/dim	128	256	512	1024	2048	4096
Running time	0.009	0.035	0.243	1.917	23.526	248.450
Work	0.017	0.129	0.971	7.637	93.431	987.196
Critical path	0.002	0.007	0.028	0.124	0.435	1.857
MFLOPS	453.732	951.008	1100.154	1119.920	730.241	553.183

rectmul/dim	128	256	512	1024	2048	4096
Running time	0.003	0.014	0.108	0.606	4.456	36.590
Work	0.003	0.025	0.280	2.138	15.556	129.474
Critical path	0.00002	0.00003	0.002	0.0007	0.070	0.050
MFLOPS	1166.690	2286.555	2474.950	3537.611	3853.949	3755.658

spacemul/dim	128	256	512	1024	2048	4096
Running time	0.008	0.020	0.097	0.500	3.555	25.783
Work	0.009	0.052	0.258	1.761	12.929	94.108
Critical path	0.001	0.004	0.0129	0.001	0.056	0.013
MFLOPS	501.190	1639.924	2761.371	4288.247	4830.626	5329.944

Matrix Multiplication (8192×8192)

matmul/nproc	4	8	16	32	64
Running time	1886.856	953.561	494.844	249.944	130.712
Work	7493.744	7575.331	7863.391	7950.240	8322.858
Critical path	6.558	8.087	8.909	14.322	22.115
MFLOPS	582.721	1153.058	2221.935	4399.031	8411.650

rectmul/nproc	4	8	16	32	64
Running time	268.757	137.131	72.831	38.172	22.727
Work	942.324	963.211	1022.434	1072.025	1249.097
Critical path	0.042	0.692	0.863	0.906	0.792
MFLOPS	4090.849	8017.454	15095.658	28801.972	48375.711

spacemul/nproc	4	8	16	32	64
Running time	221.009	96.696	63.574	31.731	19.226
Work	805.811	704.145	928.907	926.700	1124.154
Critical path	0.009	0.142	0.532	0.359	0.620
MFLOPS	4974.651	11370.052	17293.722	34648.866	57185.041

References

- Matteo Frigo, Multithreaded Programming in Cilk, invited talk at PASCO'2007. http://www.informatik.uni-trier.de/ ley/db/conf/issac/pasco2007.html
- Charles E. Leiserson. Theory of Parallel Systems, course scribe notes, 2003. http://courses.csail.mit.edu/6.895/fall03/
- Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, Pages: 212-223. June, 1998.
- Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. Journal of Parallel and Distributed Computing, 55-69, August 25, 1996.
- Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. Journal of the ACM, Vol. 46, No. 5, pp. 720-748. September 1999.
- Cilk example programs. http://supertech.csail.mit.edu/cilkImp.html