Shared Memory Parallelism in Ada: Load Balancing by Work Stealing

Jan Verschelde

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
http://www.math.uic.edu/~jan
janv@uic.edu

www.phcpack.org

Ada devroom, FOSDEM 2018, 3 February, Brussels, Belgium

Outline

1. Problem Statement
   - computing the permanent of a matrix
   - high level parallel programming

2. Multitasking in Ada
   - launching a crew of workers
   - work stealing with multitasking
   - application to polynomial system solving

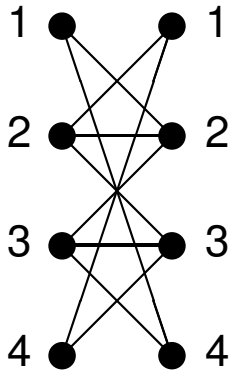
Load Balancing by Work Stealing

1. Problem Statement
   - computing the permanent of a matrix
   - high level parallel programming

2. Multitasking in Ada
   - launching a crew of workers
   - work stealing with multitasking
   - application to polynomial system solving

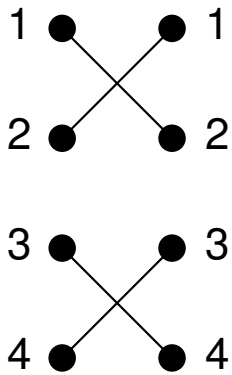
all perfect matchings in a bipartite graph

Consider the adjacency matrix of a bipartite graph:

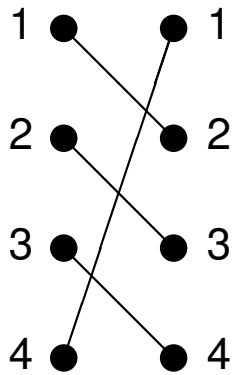


$$A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \qquad \text{perm}(A) = 5$$

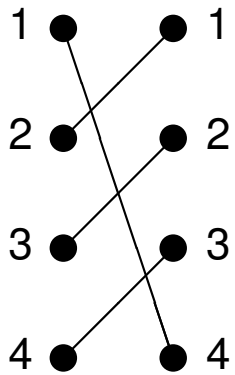
The permanent counts all perfect matchings in the graph:



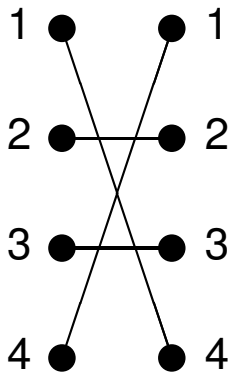
2 1 4 3



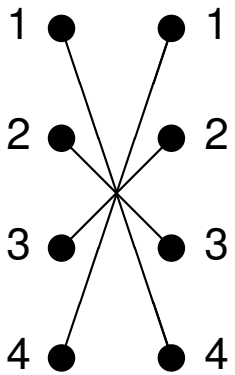
2 3 4 1



4 1 2 3



4 2 3 1



4 3 2 1

row expansions

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

We expand along the rows:

$$\begin{aligned} \mathrm{perm}(A) &= 1 \times \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} + 1 \times \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \\ &= 1 \times \left(1 \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} + 1 \times \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right) \\ &+ 1 \times \left(1 \times \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} + 1 \times \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} + 1 \times \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right) \\ &= \cdots \end{aligned}$$

computational experiments

The permanent of an n-by-n matrix A is

$$\text{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)},$$

where S_n is the set of all permutations of n numbers, $\#S_n = n!$.

On a MacBook Pro 3.1 GHz Intel Core i7,
timings on randomly generated Boolean matrices,
of dimension $n = 14, 15, 16, 17$, the CPU time in seconds:

n	time
14	1.439
15	10.419
16	58.497
17	170.828

expanding the first two rows

Consider the first two rows in the matrix A:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{matrix} 2 & 1 & \cdots \\ 2 & 3 & \cdots \\ 2 & 9 & \cdots \\ 3 & 1 & \cdots \\ 3 & 9 & \cdots \\ 4 & 1 & \cdots \\ 4 & 3 & \cdots \\ 4 & 9 & \cdots \\ 6 & 1 & \cdots \\ 6 & 3 & \cdots \\ 6 & 9 & \cdots \end{matrix}$$

At the right are the expansions of the first two rows.
Those expansions represent 11 computationally independent jobs.

Load Balancing by Work Stealing

1. Problem Statement
   - computing the permanent of a matrix
   - high level parallel programming

2. Multitasking in Ada
   - launching a crew of workers
   - work stealing with multitasking
   - application to polynomial system solving

shared memory parallel programming

Consider a parallel computation by p processors:

1. all processors share the same memory space;
2. the jobs can be computed independently.

We can work with one static queue of jobs:

- The queue is initialized with jobs.
- Jobs are popped from the front of the queue.
- Popping jobs is guarded by a semaphore.
- Idle workers pop jobs till the queue is empty.

This is the *work crew* model of multithreading.

load balancing by work stealing

In the work crew model, processors take jobs from one queue.

In work stealing, underutilized processors steal jobs:

- Every processor has its own dequeue of jobs.
  A dequeue is a double ended queue, with beginning and end.
- Jobs are appended to the end of the dequeue.

  A processor treats its own dequeue as a stack:
  - pushing new jobs to the end,
  - popping jobs from the end.
- Processors with empty job queues steal jobs from others, popping from the beginning of their dequeue.

This idea appeared first in [Burton and Sleep, 1981].
The first provably good work stealing scheduling algorithm appeared in [Blumofe and Leiserson, 1994].

Load Balancing by Work Stealing

1. Problem Statement
   - computing the permanent of a matrix
   - high level parallel programming

2. Multitasking in Ada
   - launching a crew of workers
   - work stealing with multitasking
   - application to polynomial system solving

starting worker tasks

procedure `Workers` is instantiated with a `Job` procedure, executing code based on the `id` number.

```
procedure Workers ( n : in natural ) is
   task type Worker ( id,n : natural );
   task body Worker is
   begin
     Job(id,n);
   end Worker;
   procedure Launch_Workers ( i,n : in natural ) is
     w : Worker(i,n);
   begin
     if i < n
      then Launch_Workers(i+1,n);
     end if;
   end Launch_Workers;
begin
   Launch_Workers(1,n);
end Workers;
```

managing the job queue for the work crew

On input is a list of partially selected column indices.
The job queue is then the corresponding list of pointers:
each job requires the application of recursive row expansions.

The permanent computation is then a pleasingly parallel computation:
no communication overhead during the row expansion.

Management of the job queue:

1. an idle worker requests access to the next pointer in the queue;
2. once given access, the worker takes the job and becomes busy;
3. the factor is added to the factors computed by the worker.

Dynamic load balancing works well in this way.

Source of inspiration: Gem #81: GNAT Semaphores, at
`http://www.adacore.com/adaanswers/gems/gem-81`

wall clock times in seconds on 3.1 GHz Intel Core i7

Random Boolean matrices of dimension 16 are generated.
With 2 tasks, jobs are generated expanding the first two rows:

#jobs	permanent	serial	2 tasks	speedup
39	205676452	48	26	1.85
74	398844456	108	65	1.66
58	457676445	79	44	1.79
14	96908415	16	10	1.60
64	58417614	17	9	1.88

With 4 tasks, the first 3 rows are expanded, for a finer granularity:

#jobs	permanent	serial	4 tasks	speedup
278	282852334	45	24	1.88
420	268894344	95	52	1.83
521	39106098	14	7	2.00
321	77841276	37	20	1.85
359	1394427180	236	126	1.87

wall clock times in seconds on 3.1 GHz Intel Core i7

Random Boolean matrices of dimension 16 are generated.
With 3 tasks, expanding the first 3 rows gives more jobs:

#jobs	permanent	serial	3 tasks	speedup
275	29320581	8	4	2.00
173	134237181	27	15	1.80
485	549654797	92	55	1.67
324	158044038	27	15	1.80
597	36928234	11	6	1.83

With 3 tasks, expanding only the first 2 rows gives fewer jobs:

#jobs	permanent	serial	3 tasks	speedup
50	111120492	15	8	1.88
38	116785084	44	22	2.00
39	224525956	35	18	1.94
53	67912248	9	5	1.80
66	497301012	112	56	2.00

44-core computer 2.2 GHz Intel Xeon E5-2699

On a random Boolean matrix of dimension 17,
wall clock times are measured in seconds,
jobs are generated expanding the first 3, 3, 4 rows:

#jobs	permanent		#jobs	permanent		#jobs	permanent	
314	1413427296		188	412123207		1432	1452757932	
#tasks	time	speedup	#tasks	time	speedup	#tasks	time	speedup
1	284		1	152		1	431	
2	172	1.65	2	86	1.76	2	238	1.81
4	89	3.19	4	45	3.78	4	122	3.53
8	49	5.80	8	24	6.33	8	63	6.84
16	25	11.36	16	13	11.69	16	33	13.06
32	15	18.93	32	8	19.00	32	19	22.68
64	11	25.81	64	6	25.33	64	14	30.79

Load Balancing by Work Stealing

1. Problem Statement
   - computing the permanent of a matrix
   - high level parallel programming

2. Multitasking in Ada
   - launching a crew of workers
   - work stealing with multitasking
   - application to polynomial system solving

work stealing with multitasking

The main data structure is a dequeue, with a beginning and an end.

Although only one processor appends to the dequeue,
both beginning and end must have semaphores.

- Underutilized processors pop from the beginning.
  Only one underutilized processor may pop, other underutilized processors must wait or skip to the following dequeue.
- New jobs are pushed to the end and popped from the end.
  If there is only one last job left, then the processor may have to wait for an underutilized processor.

We have a sequence of dequeues, one for every task.

application of work stealing to the permanent

Instead of generating jobs before the launching of the tasks, each task generates its own dequeue of jobs:

- If for an n-by-n matrix, we consider only the first row, then task k takes column k, for $k \leq n$.
- The permutations in the row expansions are serialized. For p tasks, the kth task takes permutation k, $k+p$, $k+2p$, etc. and computes the factors only for those permutations.
- Every task runs the same enumeration of permutations. The number of permutations in this enumeration is fixed in advance, before the launching of the tasks.

The load balancing in this static job assignment scheme then happens via work stealing. For p processors, task k starts looking at the dequeue of task $k+1$, $k+2$, etc. modulo p.

Load Balancing by Work Stealing

1. Problem Statement
   - computing the permanent of a matrix
   - high level parallel programming

2. Multitasking in Ada
   - launching a crew of workers
   - work stealing with multitasking
   - application to polynomial system solving

solving polynomial systems with PHCpack

PHCpack is a package for Polynomial Homotopy Continuation.
ACM Transactions on Mathematical Software achived version 1.0 (Ada 83) as Algorithm 795, vol. 25, no. 2, pages 251–276, 1999.

blackbox solver:

`phc -b` computes all isolated solutions of a polynomial system.

Version 2.0 was rewritten using concepts of Ada 95
and extended with arbitrary multiprecision arithmetic.

The current version is 2.3.48, last released December 2017.

Distributed under the GNU General Public License.

Public repository under version control
at `https://github.com/janverschelde/PHCpack`.

Code for this talk is in the folder `src/Tasking`.

applications to polynomial systems

- A permanent is a bound on the number of solutions.

  For example, the permanent gives the number of totally mixed Nash equilibria for any number of players with two pure strategies.

- We applied work stealing to intersections of polyhedral cones.

  This work is published in the Proceedings of PASCO 2017.