# Distributed Memory Programming (Heterogeneity)

ICS632: Principles of High Performance Computing

Henri Casanova (henric@hawaii.edu)

Fall 2015

## Heterogeneous Platforms

- So far we've assumed that all processors are identical
  - Representative for many parallel platforms
- But many platforms are heterogeneous
  - A cluster for which new nodes are purchased and installed
  - Several homogeneous clusters connected to each other
  - A parallel platform designed from the get-go to be heterogeneous
  - Nothing is truly homogeneous anyway...
- Consequently, we need to design parallel programs that achieve high performance on heterogeneous platforms
- The main issue is Load Balancing

## Heterogeneous Load Balancing

- There is a large literature on load-balancing for heterogeneous platforms
    - CPUs of various speeds, CPUs and GPUs, heterogeneous cores, etc.
    - Network heterogeneity is also studied
- We'll look at static load balancing
    - Before the application starts we know how much work we give to each processor
    - We'll talk about dynamic load balancing in another set of lecture notes
- Let's revisit some of the algorithms we have already discussed and see how to make them heterogeneous-friendly
    - We assume a homogeneous network

## Straightforward Static Task Allocation

- $P$ processors $(p_1, \ldots, p_P)$, $N$ identical tasks to do
- Processor $p_i$ processes a task in $t_i$ seconds
- $c_i$: the number of tasks processed by processor $p_i$, which is unknown
- Rational solution:

$$c_i = \frac{\frac{1}{t_i}}{\sum_{k=1}^{P} \frac{1}{t_k}} \times N$$

- $\frac{1}{t_i}$ is the *compute speed* of processor $i$
- We simply assign tasks to processors proportionally to their compute speeds
- Problem: $c_i$ may not be an integer

## Optimal Integral Task Allocation

- Initialize with rational values, rounded down:

$$\forall i \; c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{k=1}^{P} \frac{1}{t_k}} \times N \right\rfloor$$

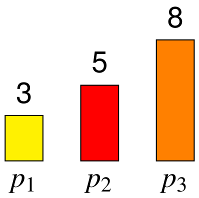
- Complete with left-over tasks

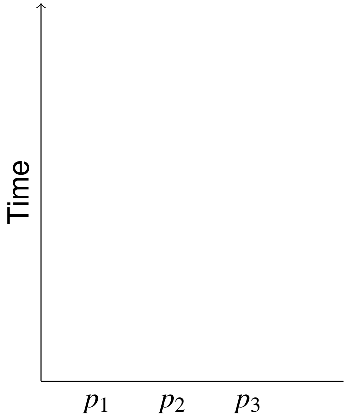
```
// While there remain tasks to be assigned
while (c[1] + ... + c[p] < N) {
  // Find the processor that would finish the earliest if  assigned one extra task
  found = 1;
  for (k = 2; k <= p; i++) {
    if (t[k] * (c[k] + 1) < t[found] * (c[found] + 1) {
      found = k;
    }
  }
  // Assign one extra task to that processor
  c[found]++;
}
```

## Example

3 processors, 11 tasks

$t_1 = 3$, $t_2 = 5$, $t_3 = 8$

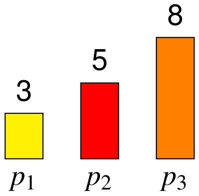


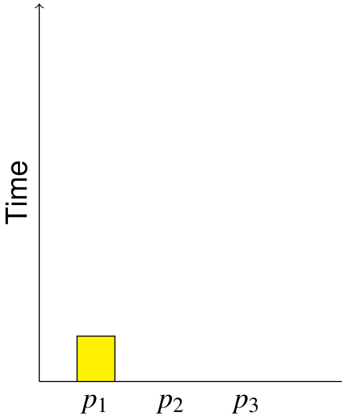


## Example

3 processors, 11 tasks

$t_1 = 3,\ t_2 = 5,\ t_3 = 8$

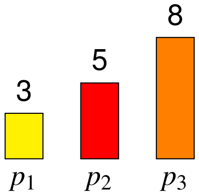


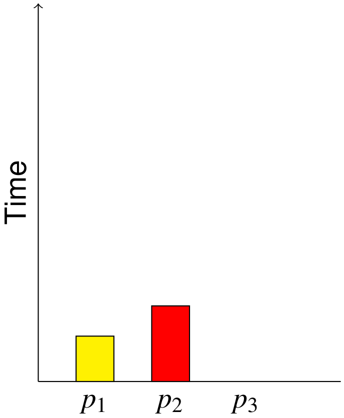


## Example

3 processors, 11 tasks

$t_1 = 3$, $t_2 = 5$, $t_3 = 8$

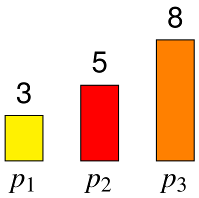


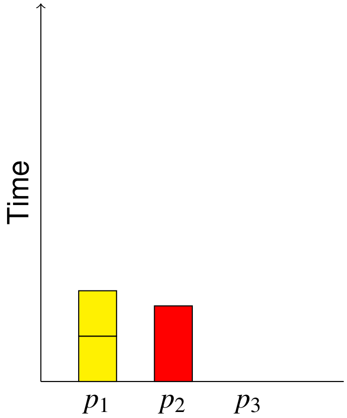


## Example

3 processors, 11 tasks

$t_1 = 3, t_2 = 5, t_3 = 8$

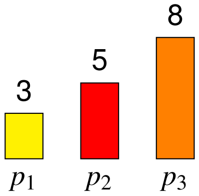


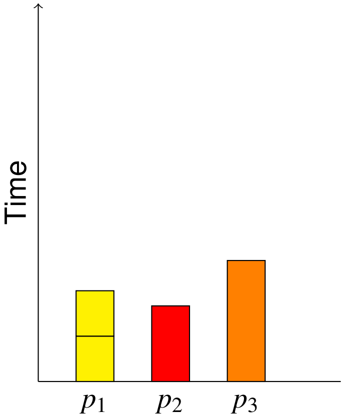


## Example

3 processors, 11 tasks

$t_1 = 3,\ t_2 = 5,\ t_3 = 8$

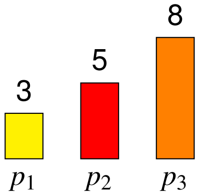


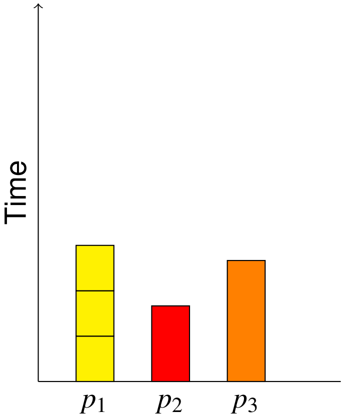


## Example

3 processors, 11 tasks

$t_1 = 3$, $t_2 = 5$, $t_3 = 8$

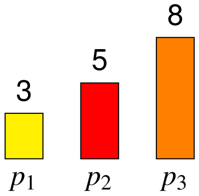


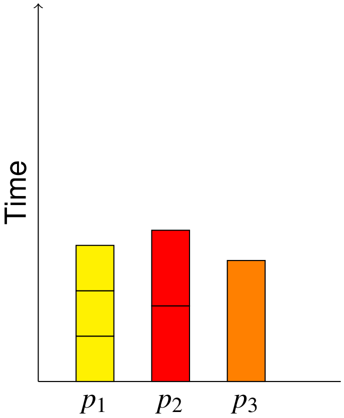


## Example

3 processors, 11 tasks

$t_1 = 3, t_2 = 5, t_3 = 8$

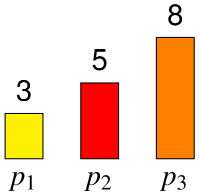


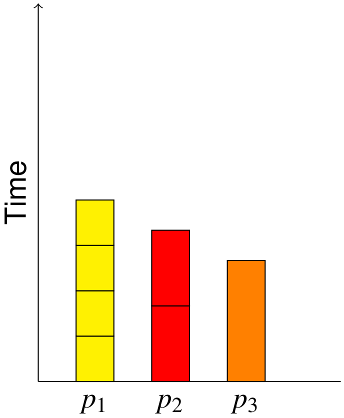


## Example

3 processors, 11 tasks

$t_1 = 3, t_2 = 5, t_3 = 8$

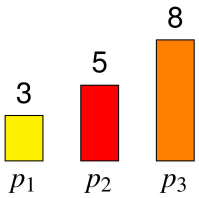


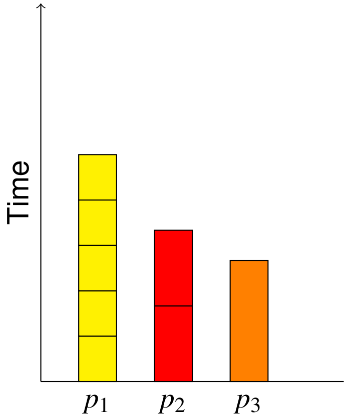


## Example

3 processors, 11 tasks

$t_1 = 3, t_2 = 5, t_3 = 8$

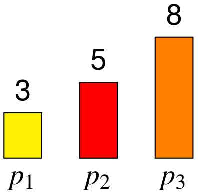


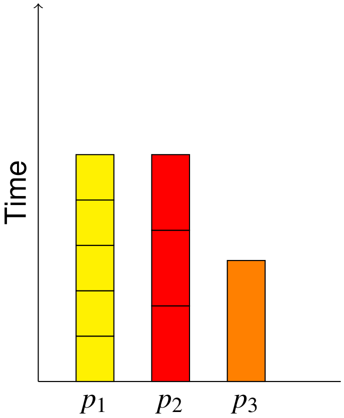


## Example

3 processors, 11 tasks

$t_1 = 3, t_2 = 5, t_3 = 8$

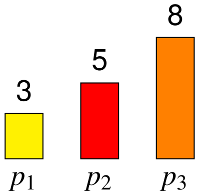


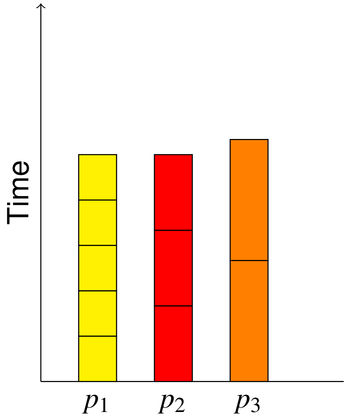


## Example

3 processors, 11 tasks

$t_1 = 3, t_2 = 5, t_3 = 8$

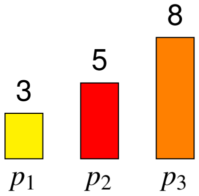


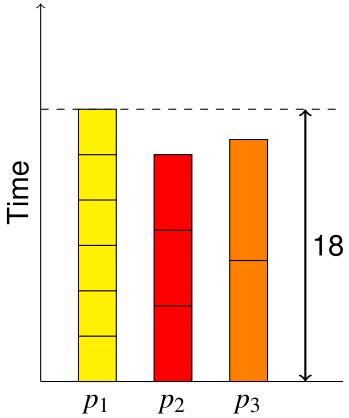


## Example

3 processors, 11 tasks

$t_1 = 3, t_2 = 5, t_3 = 8$



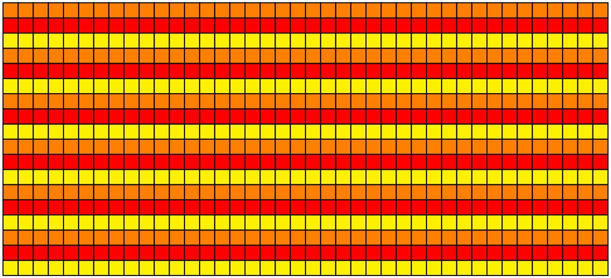


## Incremental Enumeration

- The algorithm can be modified easily to produce incremental optimal solutions:
  - $N = 1$: $p_1$
  - $N = 2$: $p_1, p_2$
  - $N = 3$: $p_1, p_2, p_1$
  - $N = 4$: $p_1, p_2, p_1, p_3$
  - $N = 5$: $p_1, p_2, p_1, p_3, p_1$
  - $N = 6$: $p_1, p_2, p_1, p_3, p_1, p_2$
  - $N = 7$: $p_1, p_2, p_1, p_3, p_1, p_2, p_1$
  - ...
- We'll see how this can come in handy later in these lecture notes

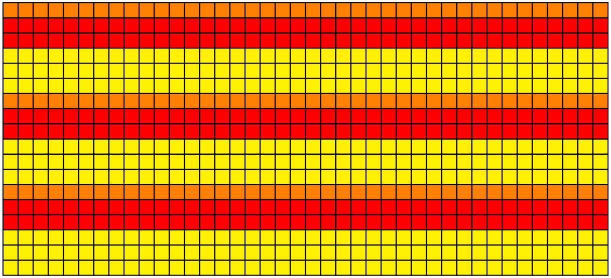
## Stencil Application

- Recall our 1-D cyclic data distribution for a stencil application



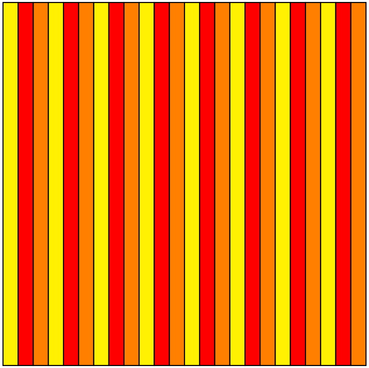
## Stencil Application

- For a homogeneous platform our simple algorithm gives us a data distribution
  - Simple in principle, requires careful coding



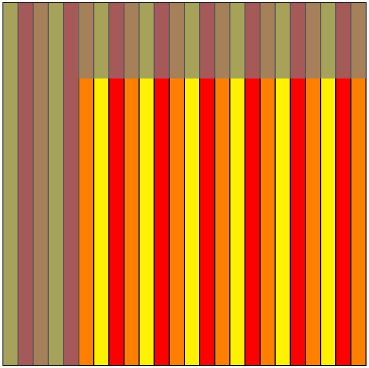
## LU Factorization

- Cyclic 1-D data distribution



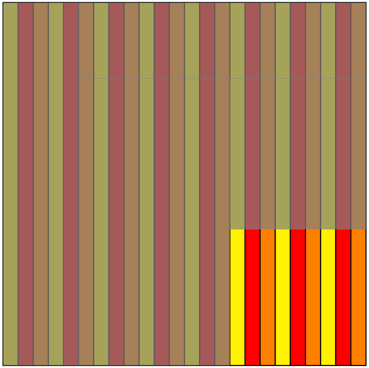
## LU Factorization

- Cyclic 1-D data distribution



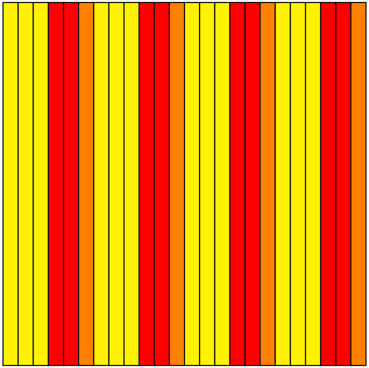
## LU Factorization

- Cyclic 1-D data distribution



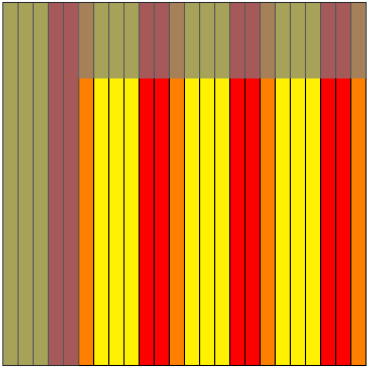
## LU Factorization

- Cyclic heterogeneous distribution?



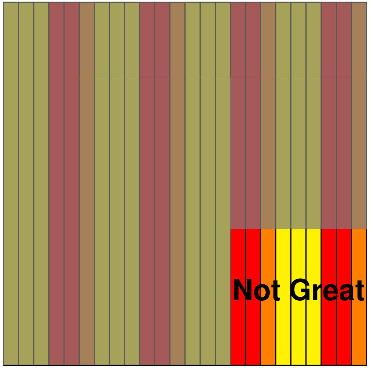
## LU Factorization

- Cyclic heterogeneous distribution?



## LU Factorization

- Cyclic heterogeneous distribution?

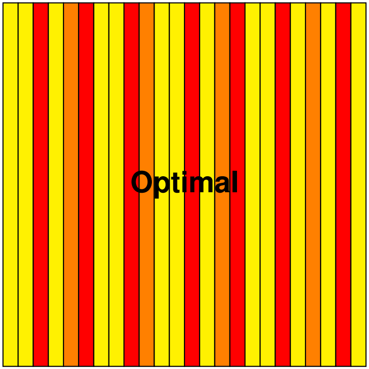


## LU Factorization

- Recall the incremental solution...
  - $N = 1$: $p_1$
  - $N = 2$: $p_1, p_2$
  - $N = 3$: $p_1, p_2, p_1$
  - $N = 4$: $p_1, p_2, p_1, p_3$
  - $N = 5$: $p_1, p_2, p_1, p_3, p_1$
  - $N = 6$: $p_1, p_2, p_1, p_3, p_1, p_2$
  - $N = 7$: $p_1, p_2, p_1, p_3, p_1, p_2, p_1$
  - $N = 8$: $p_1, p_2, p_1, p_3, p_1, p_2, p_1, p_1$
  - $N = 9$: $p_1, p_2, p_1, p_3, p_1, p_2, p_1, p_1, p_2$
  - ...
- If we reverse this solution:
  - $N = 9$: $p_2, p_1, p_1, p_2, p_1, p_3, p_1, p_2, p_1$
- Then each *suffix* is an optimal distribution!!
- This give us our optimal data distribution for LU...

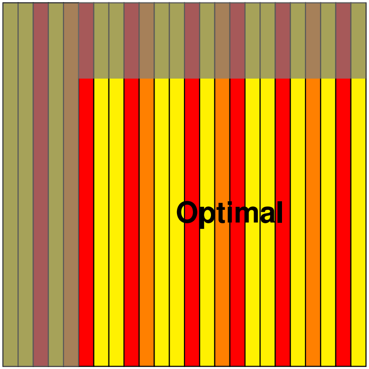
## LU Factorization

- Optimal data distribution



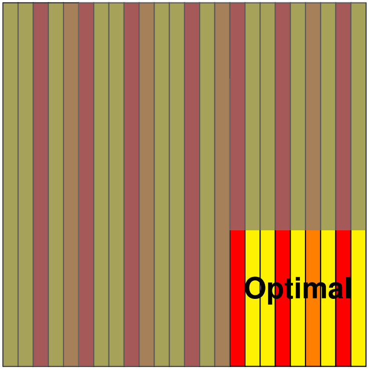
## LU Factorization

- Optimal data distribution



## LU Factorization

- Optimal data distribution



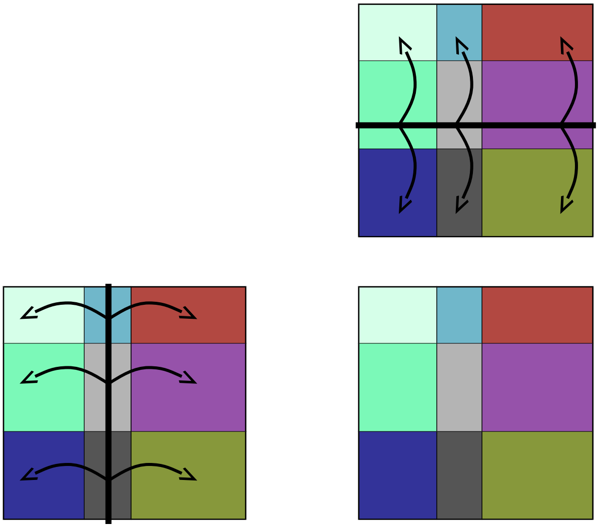
## 2-D Data Distribution for Matrix Multiplication

- We saw simple 2-D data distributions for matrix multiplication
- Can we adapt these distributions to the heterogeneous case?

## Outer-product Algorithm

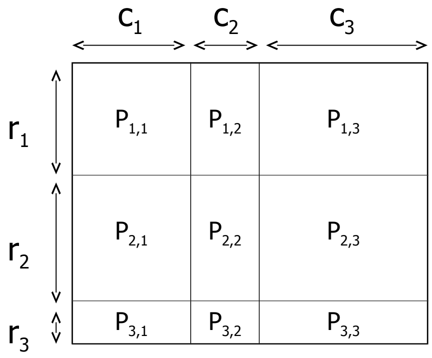
- Proceeds in k=1,...,n steps
- *Horizontal broadcasts*: $P_{i,k}$, for all i=1,...,p, broadcasts $a_{ik}$ to processors in its processor row
- *Vertical broadcasts*: $P_{k,j}$, for all j=1,...,q, broadcasts $a_{kj}$ to processors in its processor column
- *Independent computations*: processor $P_{i,j}$ can update $c_{ij} = c_{ij} + a_{ik} \times a_{kj}$

## Outer-product Algorithm



## Load-balancing

- Let $t_{i,j}$ be the cycle time of processor $P_{i,j}$
- We assign to processor $P_{i,j}$ a rectangle of size $r_i$ x $c_j$



## Load-balancing

- First, let us note that it is not always possible to achieve perfect load-balacing
  - There are some theorems that show that it's only possible if the processor grid matrix, with processor cycle times, $t_{ij}$, put in their spot is of rank 1
- Each processor computes for $r_i \times c_j \times t_{ij}$ time
- Therefore, the total execution time is

  $T = \max_{i,j} \{r_i \times c_j \times t_{ij}\}$

## Load-balancing as optimization

- Load-balancing can be expressed as a constrained minimization problem
- minimize $\max_{i,j} \{r_i \times c_j \times t_{ij}\}$
- with the constraints

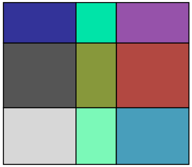
$$\sum_{i=1}^{p} r_i = n \qquad \sum_{j=1}^{p} c_j = n$$

## Load-balancing as optimization

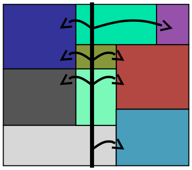
- The load-balancing problem is in fact much more complex
  - One can place processors in any place of the processor grid
  - One must look for the optimal given all possible arrangements of processors in the grid (and thus solve an exponential number of the the optimization problem defined on the previous slide)
- The load-balancing problem is NP-hard
- Complex proof
- A few (non-guaranteed) heuristics have been developed
  - they are quite complex

## “Free” 2-D distribution

- So far we’ve looked at things that looked like this



- But how about?



## Free 2-D distribution

- Each rectangle must have a surface that's proportional to the processor speed
- One can "play" with the width and the height of the rectangles to try to minimize communication costs
  - A communication involves sending/receiving rectangles' half-perimeters
  - One must minimize the sum of the half-perimeters if communications happen sequentially
  - One must minimize the maximum of the half-perimeters if communications happen in parallel

## Problem Formulation

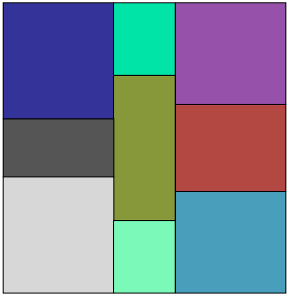
- Let us consider p numbers $s_1, \ldots, s_p$ such that $s_1+\ldots+s_p = 1$
  - we just normalize the sum of the processors' cycle times so that they all sum to 1
- Find a partition of the unit square in p rectangles with area $s_i$, and with shape $h_i x v_i$ such that

  $h_1+v_1 + h_2+v_2 + \ldots + h_p+v_p$

  is minimized.
- This problem is NP-hard

## Guaranteed Heuristic

- There is a guaranteed heuristic (that is within some fixed factor of the optimal)
  - non-trivial (look in the Section 6.3 if you're curious)
- It only works with processor columns



## Heterogeneity Challenges

- Heterogeneity makes parallel computing more difficult
  - There are some NP-hard problems
  - But there are also some clever optimal solutions
- And there are tons of “heterogeneities"
  - Heterogeneity of network paths
  - Heterogeneity of memory access latencies
  - Heterogeneity of efficiencies
    - e.g., some part of the computation is better off on the CPU, some other part is better off on the GPU
- On top of it, more and more applications have non-deterministic workloads (i.e., what needs to be computed depends on the data)
- Computing a good static work allocation is just hard

## Static View of the World

- Everything we've talked about in these lecture notes so far is extremely **static**:
    - We figure out ahead of time what data each processor will hold
    - We figure out ahead of time what data each processor will communicate
    - We figure out ahead of time what computation each processor will perform
- As systems become wilder (heterogeneous, crazy topologies, GPUs, accelerators) and applications more complicated, this just becomes too difficult
- So these lectures not are a bit "the still relevant past"

## Dynamic view of the World

- A simple way to abstract a parralle application is:
  - It's a bunch of tasks with data dependencies
  - Some tasks can be created at runtime
  - Tasks are assigned to processors dynamically
    - Processors can even "steal" tasks from each other
  - But data locality is not ignored
    - If I already hold the data for a task, I should probably be the one computing that task
- This is the way state-of-the-art parallel libraries have begun being implemented!
- Let's skim the *annotated DAGuE paper* (intro + results) on the course's Web site
- This leads us to a fundamental topic: *Scheduling*