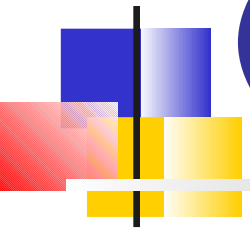
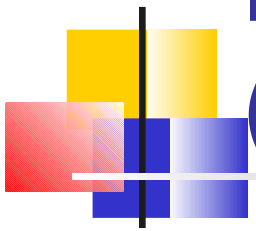


Principles of High Performance Computing (ICS 632)

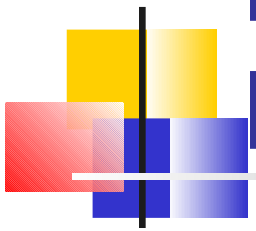
Heterogeneous Parallel Computing



Heterogeneous Platforms (Ch. 6)

- So far we've only talked about platforms in which all processors/nodes are identical
  - representative of most supercomputers
- Clusters often "end up" heterogeneous
  - Built as homogeneous
  - New nodes are added and have faster CPUs
    - A cluster that stays up 3 years will likely have several generations of processors in it
- Network of workstations are heterogeneous
  - couple together all machines in your lab to run a parallel applications (became popular with PVM)
- "Grids"
  - couple together different clusters, supercomputers, and workstations

➔ It is important to develop parallel applications that can leverage heterogeneous platforms



Heterogeneous Load Balancing

- There is an impressively large literature on load-balancing for heterogeneous platforms
- In this lecture we're looking only at "static" load balancing
  - before execution we know how much load is given to each processor
  - e.g., as opposed to some dynamic algorithm that assigns work to processors when they're "ready"
    - We'll come back to that idea when we talk about scheduling
- We will look at:
  - 2 simple load-balancing algorithms
  - application to our 1-D stencil application
  - application to the 1-D distributed LU factorization
  - discussion of load-balancing for 2-D data distributions
- We assume homogeneous network and heterogeneous compute nodes in this lecture

Static task allocation (Sec. 6.1.1/2)

- Let's consider p processors
- Let t_1,..., t_p be the "cycle times" of the processors
  - i.e., time to process one elemental unit of computation (work units) for the application (T_{comp})
- Let B be the number of (identical) work units
- Let c_1,..., c_p be the number of work units processed by each processor

  $c_1 + ... + c_p = B$

- Perfect load balancing happens if

  $c_i \times t_i$ is constant

or

$$c_i = \frac{\frac{1}{t_i}}{\sum_{k=1}^{p} \frac{1}{t_k}} \times B$$

computing speed

Static task allocation

- if B is a multiple of

$$\mathrm{lcm}(t_1, t_2, ..., t_p) \times \sum_{i=1}^{p} \frac{1}{t_i}$$

  then we can have perfect load balancing
- But in general, the formula for c_i does not give an integer solution
- There is a simple algorithm that give the **optimal** (integer) allocation of work units to processors in $O(p^2)$

Simple Algorithm

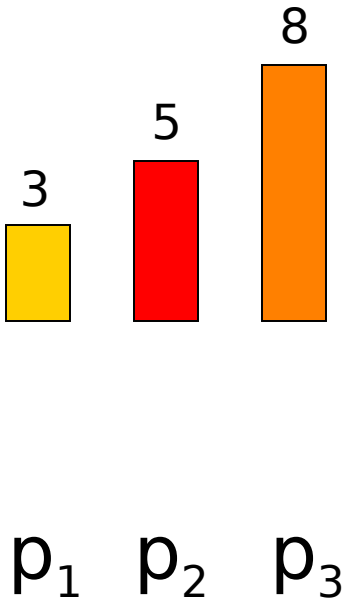
```
// initialize with fractional values
// rounded down
For i = 1 to p
```

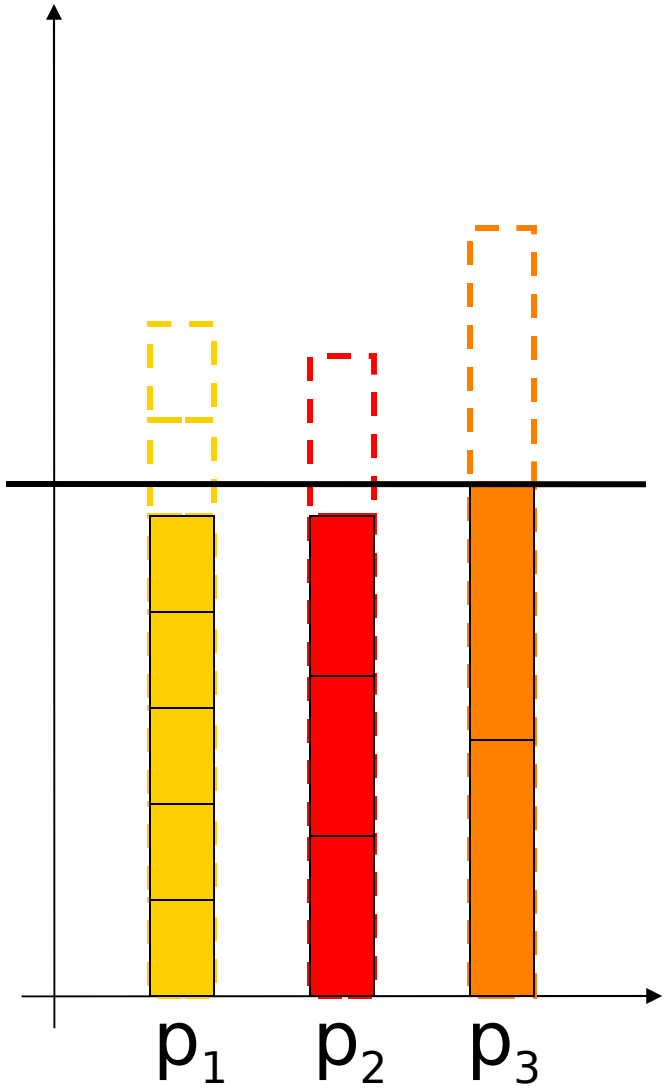
$$c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times B \right\rfloor$$

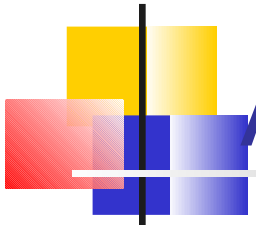
```
// iteratively increase the c_i values
while (c_1 + ... + c_p < B)
   find k in {1,...,p} such that
     t_k(c_k + 1) = min {t_i(c_i + 1)}
   c_k = c_k + 1
```

Simple Algorithm

- 3 processors
- 10 work units





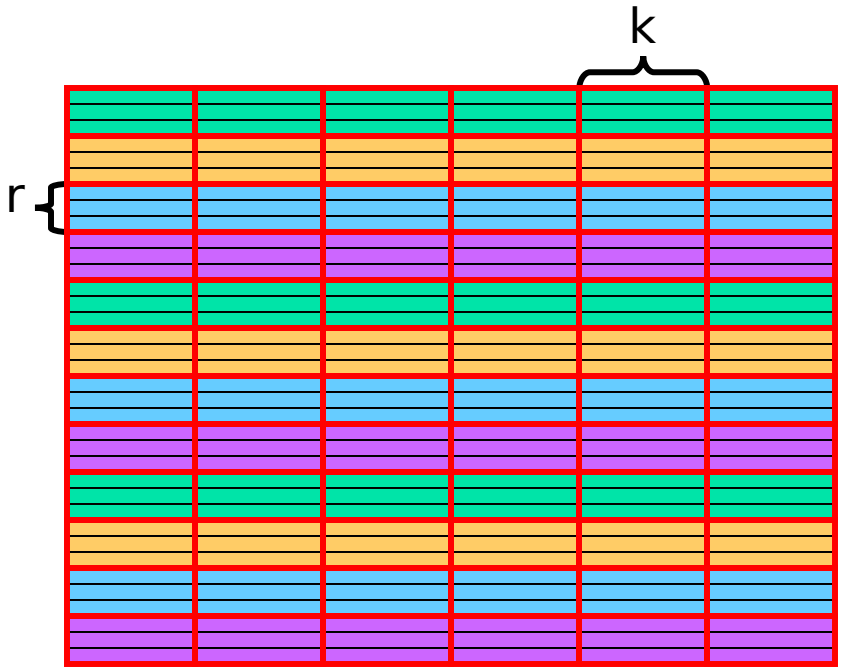


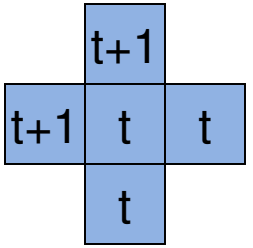
An incremental algorithm

- Note that the previous algorithm can be modified slightly to record the optimal allocation for B=1, B=2, etc...
- One can write the result as a list of processors
  - B=1: P_1
  - B=2: P_1, P_2
  - B=3: P_1, P_2, P_1
  - B=4: P_1, P_2, P_1, P_3
  - B=5: P_1, P_2, P_1, P_3, P_1
  - B=6: P_1, P_2, P_1, P_3, P_1, P_2
  - B=7: P_1, P_2, P_1, P_3, P_1, P_2, P_1
  - etc.
- We will see how this comes in handy for some load-balancing problems (e.g., LU factorization)

Stencil Application (Sec. 6.1.3)

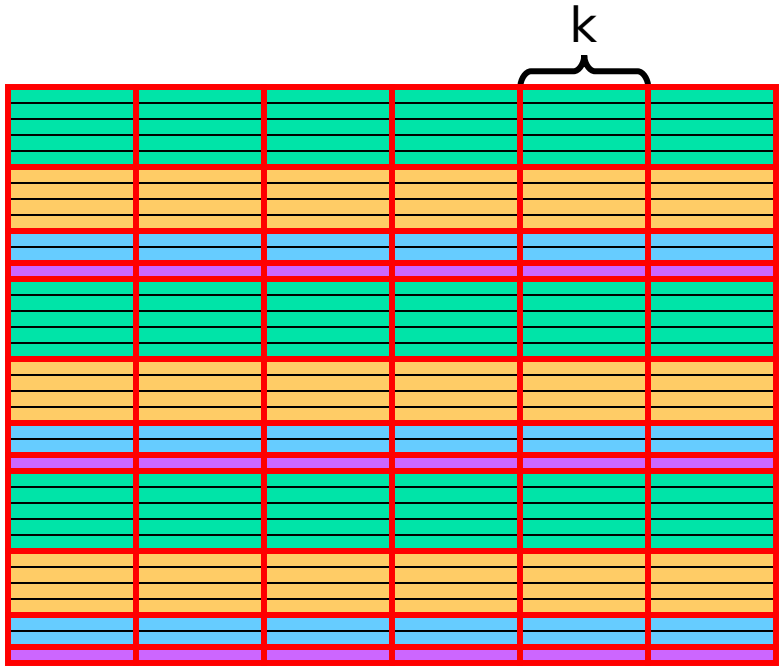
- 4 Processors
- Each processor handles many r x k blocks
- What if the processors are heterogeneous?

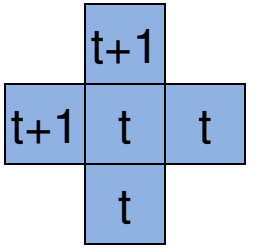




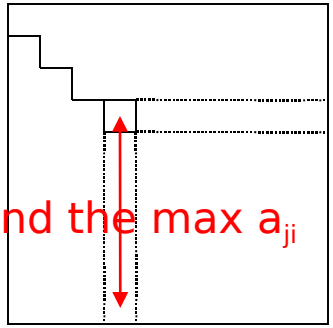
Simple Load Balancing

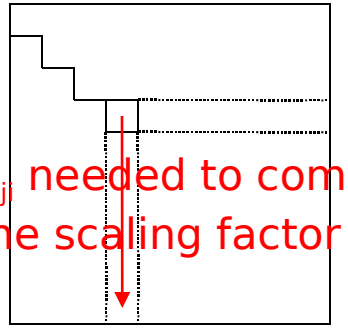
- Just give rows to processors proportionally to their speed
- Still in a cyclic pattern
- Should have perfect efficiency
- But there could of course be the usual notions of rounding off
  - e.g., what if a processor is 1.0001 faster than another?

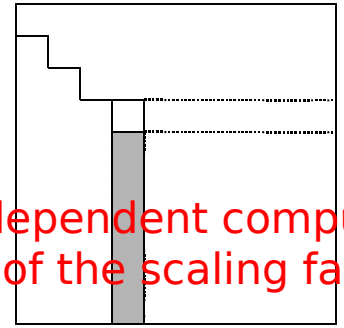




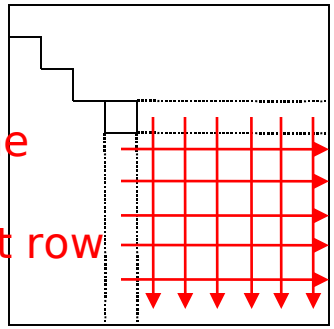
LU Decomposition

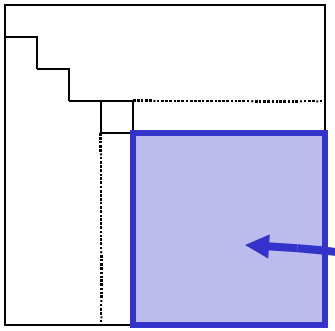






Every update needs the scaling factor and the element from the pivot row



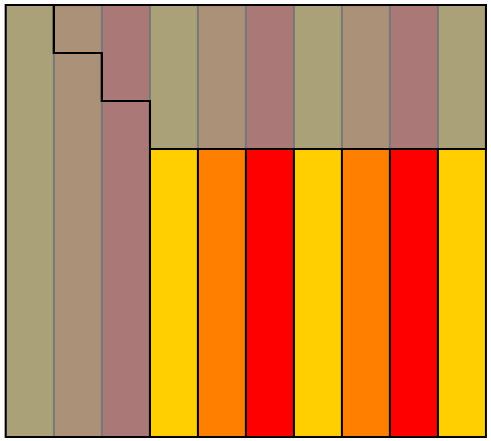


Independent computations

requires load-balancing

Load-balancing (Sect. 6.1.4)

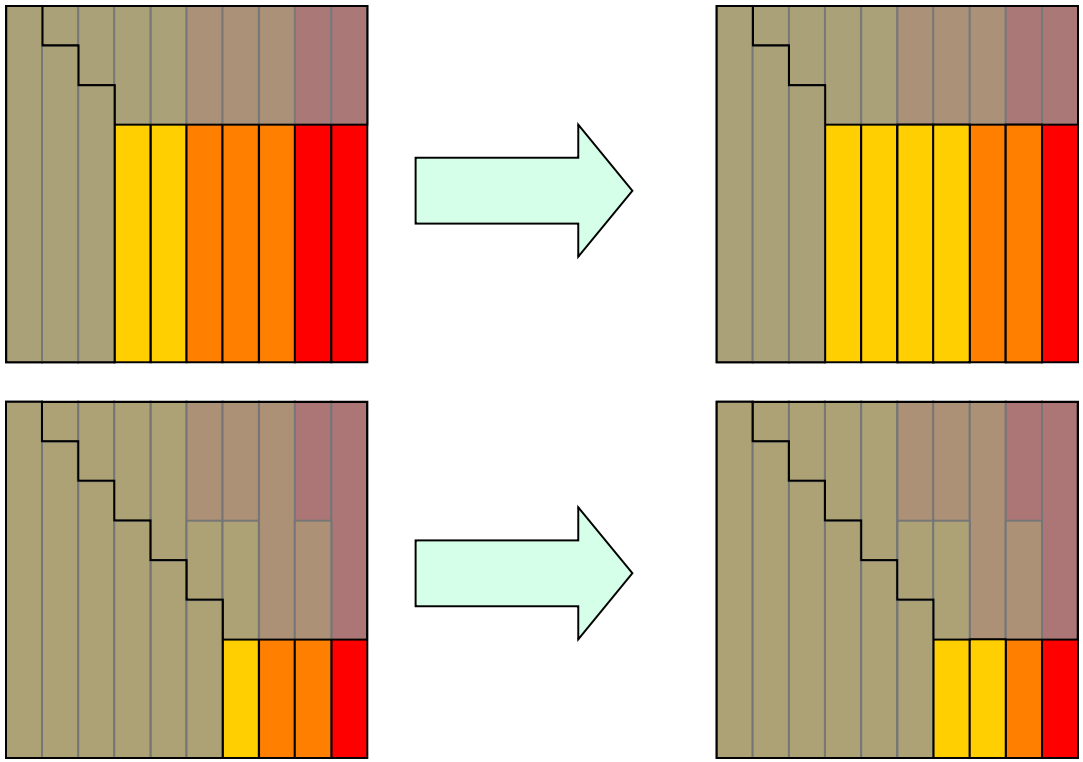
- Our original cyclic distribution doesn't work well in a heterogeneous setting



At each step all processors have the same amount of work to do

Rebalancing at each step?

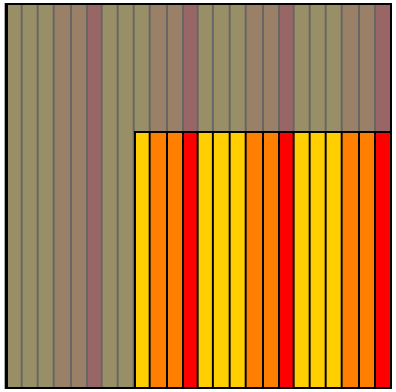
- Start with a non-cyclic distribution
- At each (or every k) steps, rebalance



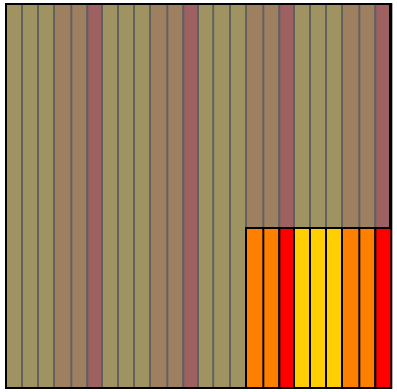
Redistribution is expensive in terms of communications

Cyclic and non-uniform

- Just do what we did for the stencil application



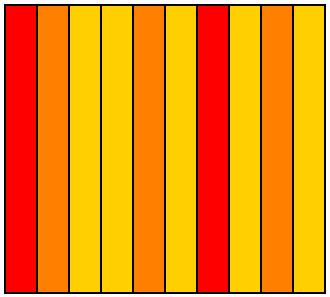
Ok...



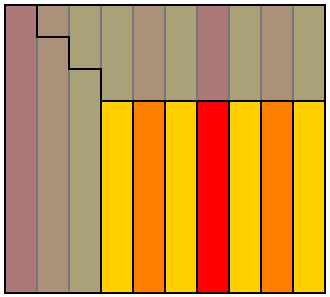
But not great

Load-balancing

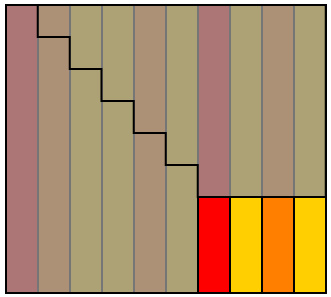
- Use the distribution obtained with the incremental algorithm we saw before, reversed
  - B=10: P_1, P_2, P_1, P_3, P_1, P_2, P_1, P_1, P_2, P_3



optimal load-balancing for 10 columns

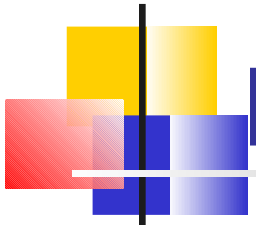


optimal load-balancing for 7 columns



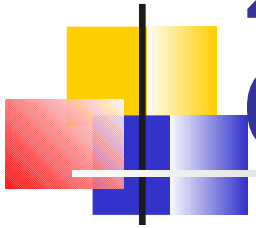
optimal load-balancing for 4 columns

. . .



Load-balancing

- Of course, this should be done for blocks of columns, and not individual columns
- Also, should be done in a “motif” that spans some number of columns (B=10 in our example) and is repeated cyclically throughout the matrix
  - provided that B is large enough, we get a good approximation of the optimal load-balance



2-D Data Distributions (Sec 6.2)

- What we've seen so far works well for 1-D data distributions
  - use the simple algorithm
  - use the allocation pattern over block in a cyclic fashion
- We have seen that a 2-D distribution is what's most appropriate, for instance for matrix multiplication
- We use matrix multiplication as our driving example

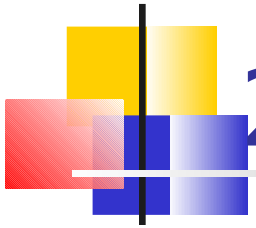
2-D Matrix Multiplication

- C = A x B
- Let us assume that we have a pxp processor grid, and that p=q=n
  - all 3 matrices are distributed identically

$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

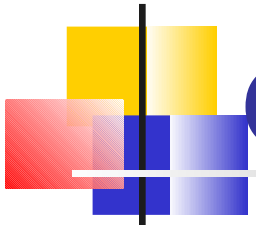
Processor Grid

$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,1}$	$A_{3,2}$	$A_{3,3}$



2-D Matrix Multiplication

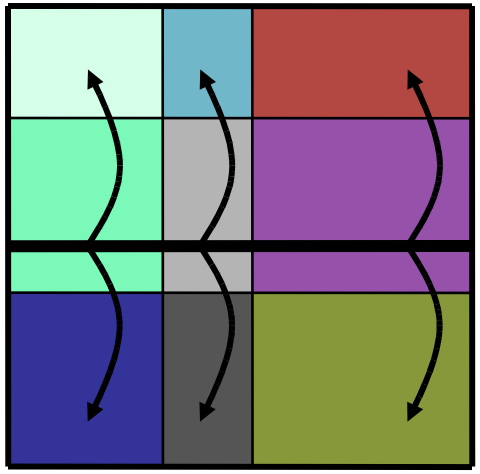
- We have seen 3 algorithms to do a matrix multiplication (Cannon, Fox, Snyder)
  - Pretty difficult to generalize them for a heterogeneous platform
- The outer product is much simpler, and thus easier to adapt and modify
- Let’s look at the outer product algorithm on a heterogeneous 2-D distribution

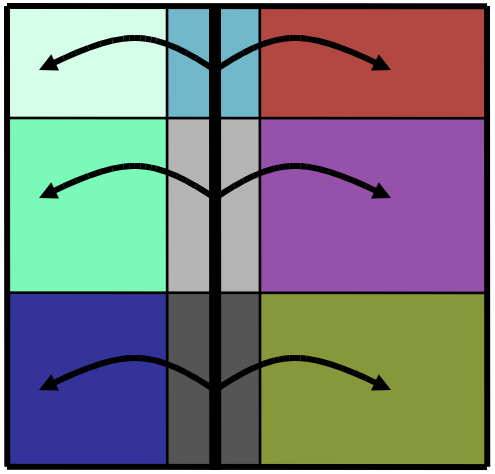


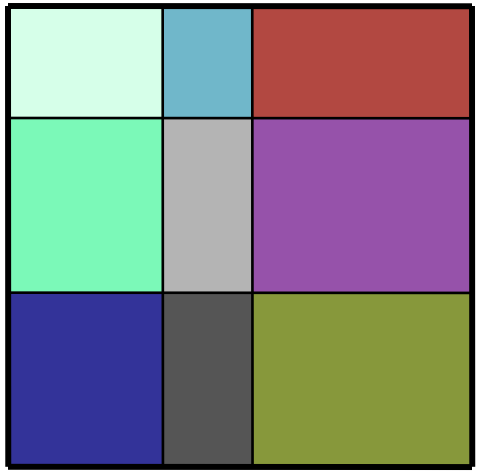
Outer-product Algorithm

- 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

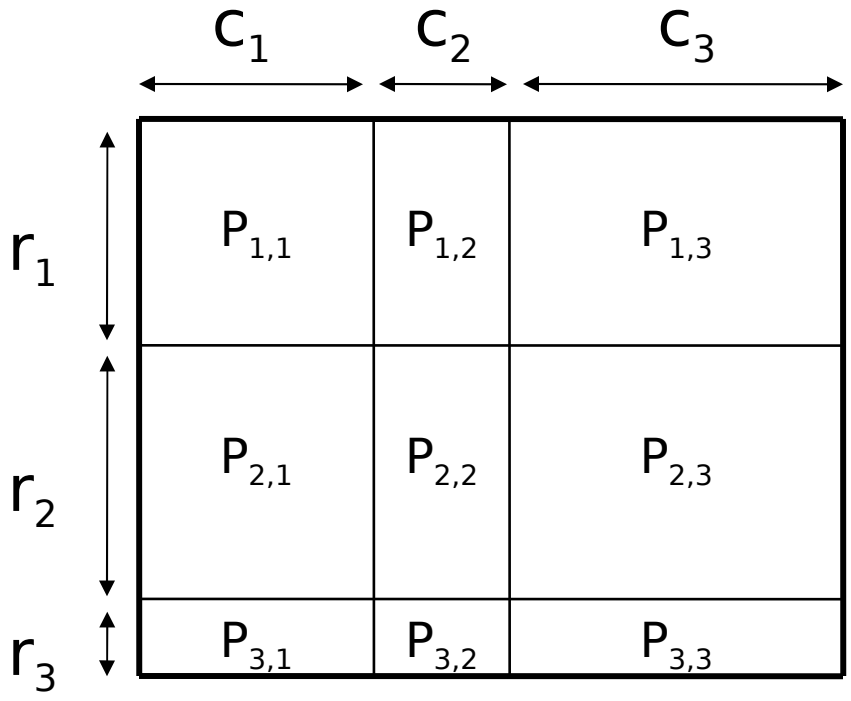


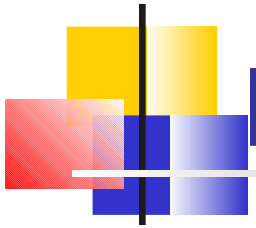




Load-balancing

- 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

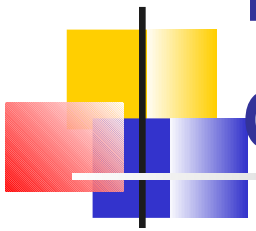
- First, let us note that it is not always possible to achieve perfect load-balacing
  - There are some theorems that show that it's only possible if the processor grid matrix, with processor cycle times, t_{ij}, put in their spot is of rank 1
- Each processor computes for $r_i \times c_j \times t_{ij}$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

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

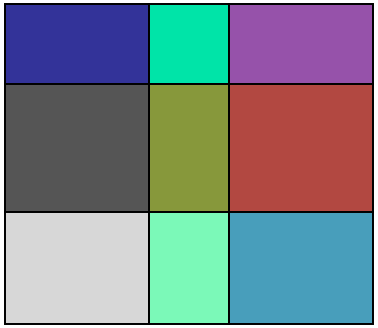


Load-balancing as optimization

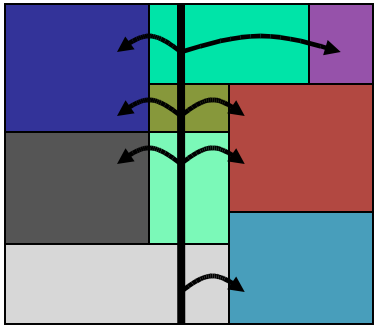
- The load-balancing problem is in fact much more complex
  - One can place processors in any place of the processor grid
  - One must look for the optimal given all possible arrangements of processors in the grid (and thus solve an exponential number of the the optimization problem defined on the previous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

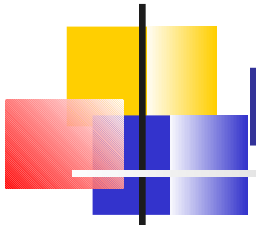
“Free” 2-D distribution

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



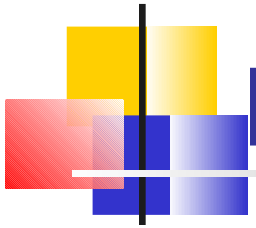
- But how about?





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

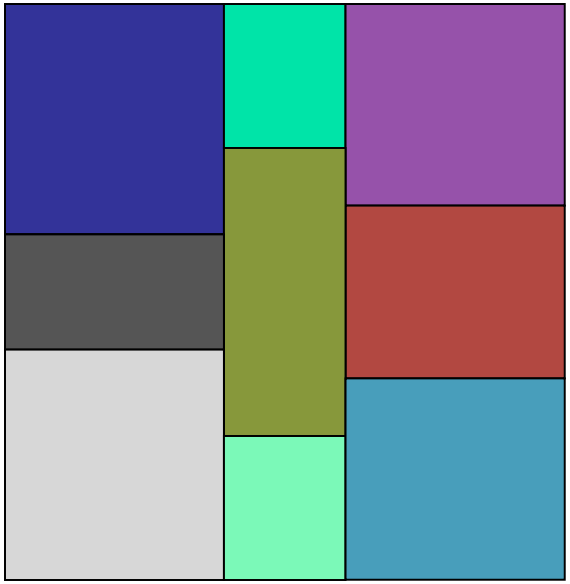
- Let us consider p numbers s_1, …, s_p such that $s_1+\ldots+s_p = 1$
  - we just normalize the sum of the processors' cycle times so that they all sum to 1
- Find a partition of the unit square in p rectangles with area s_i, and with shape $h_i x v_i$ such that

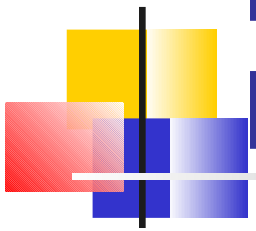
$$h_1+v_1 + h_2+v_2 + \ldots + h_p+v_p$$

  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





Heterogeneous Load-Balancing

- This all we're going to say about heterogeneous load balancing
- We'll talk more about heterogeneity when we talk about scheduling

- The terms "scheduling" and "load balancing" are often confused
- In the class, when we say "load-balancing" we mean a static data distribution that matches amount of work to processors capabilities
- When we say "scheduling" we mean that there is a notion of timing, sequencing when assigning work to processors (as we shall see in an upcoming lecture)