# CME 213

## Spring 2012-2013

Eric Darve

Stanford University

# OpenMP Master and Synchronization Constructs

## SYNCHRONIZATION CONSTRUCTS

- Several constructs are available.
- We won't go into all the details.

# Master

```
#pragma omp master
```

Definition: structured block is executed by the master thread of the team (i.e., thread ID 0).

# Critical

```
#pragma omp critical
```

Definition: the execution of the associated structured block is restricted to a single thread at a time.

## Barrier

```
#pragma omp barrier
```

Definition: explicit barrier. Threads wait until all threads reach the construct and then all can resume execution.

# ATOMIC

```
#pragma omp atomic [read | write | update | capture]
```

Definition: ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values.

Typical application:

```
int main() {
   int count = 0;
   #pragma omp parallel
   {
      #pragma omp atomic
      count++;
      // update is implied when no clause is present
   }
   printf("Number of threads: %d\n", count);
}
```

# Data sharing attributes

## THE SIMPLE VERSION

- I am explaining the "simple" version.
- Data sharing attributes. There are several options. We focus on:
  - **`shared`**: variable refers to the same block of memory for all threads
  - **`private`**: variable refers to a different block of memory for each thread
  - **`firstprivate`**: private variable but each copy is initialized with the value that the corresponding original item has when the construct is encountered

- There are three cases:
  1. pre-determined
  2. explicitly determined
  3. implicitly determined

# Pre-determined

- **Pre-determined sharing attributes (i.e., these cannot be modified):**
  - the user cannot changed the sharing attribute.
  - the attribute is determined by a rule.

- **Rules:**
  - Variables declared inside the parallel region are private ("variables with automatic storage duration declared inside the construct").
  - Static data members are shared.
  - A loop variable in a **parallel** loop is private.
  - A `const` variable is shared.

# Explicitly/Implicitly

- Explicitly determined attributes: use **`shared()`** or **`private()`**.
- **`default(none)`** or **`default(shared)`** can be used to explicitly specify a default.

- Implicitly determined attributes: basically the fall back case if not pre-determined or explicitly determined:
  - In a parallel construct, variable is shared.
  - In a task construct, variable is firstprivate.

# SPECIAL DATA-SHARING CLAUSE: REDUCTION

```
#include <omp.h>
#include <stdio.h>

int main () {

    const int n = 100;
    float a[100], b[100], result;

    /* Some initializations */
    result = 0.0;
    for (int i=0; i < n; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }

#pragma omp parallel for reduction(+:result)
    for (int i=0; i < n; i++)
        result += a[i] * b[i];

    printf("Final result= %f\n",result);
}
```

# TECHNICAL DEFINITION

**`reduction (op: list)`**

- **`op`** can be: **`{+, -, *, &, ^, |, &&, ||}`**
- For each of the variables in **`list`**, a private copy is created for each thread.
- The private copies are initialized to the neutral element (zero) of the operation **`op`** and can be updated by the owning thread.
- At the end of the region, the local values of the reduction variables are combined according to the operator **`op`** and the result of the reduction is written into the original shared variable.

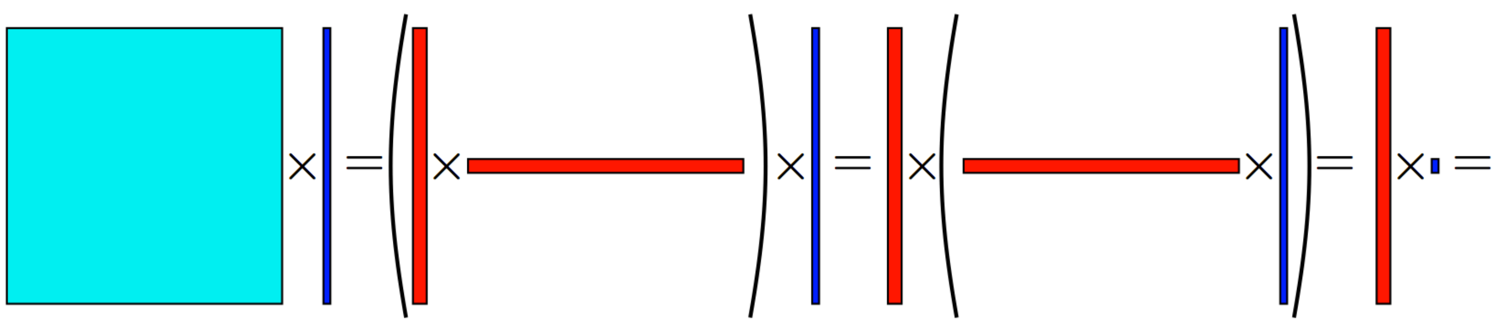
# Fast Multipole Method

# Fast Multipole Method

- A technique to calculate matrix vector products with O($N$) computational cost:

$$\phi_i = \sum_{j=1}^{N} K(\vec{r}_i, \vec{r}_j) q_j \quad j \in \{1, 2, \ldots, N\}$$

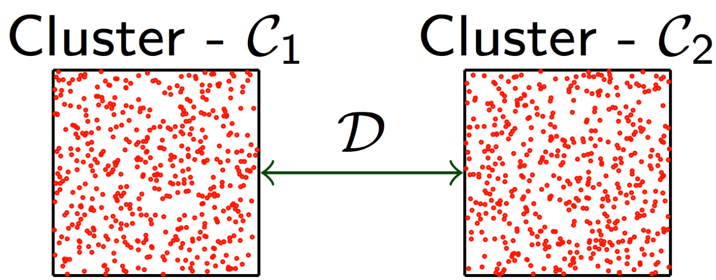
- Method is based on approximating blocks in the matrix $K(r_i, r_j)$ $K(\vec{r}_i, \vec{r}_j)$ low-rank matrix:



Example of matrix-vector product with low-rank matrix

## Finding low-rank blocks

- For most kernels $K$ in engineering applications, if we consider interacting points that belong to well-separated clusters, the interaction is approximately low-rank:

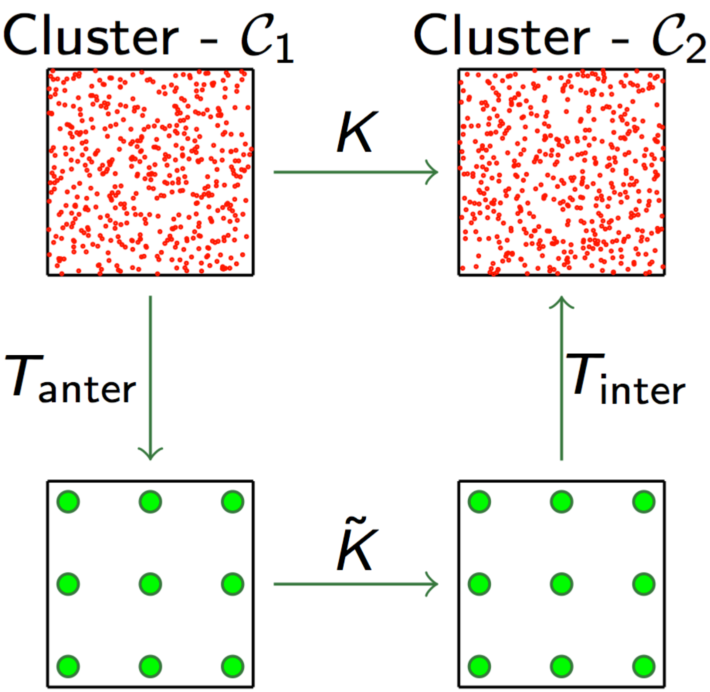



- **Well-separated criterion:** interaction is low-rank if the clusters are separated by at least $aR$, $R$ radius of clusters, with $0 < a$, e.g., $a$=0.87.

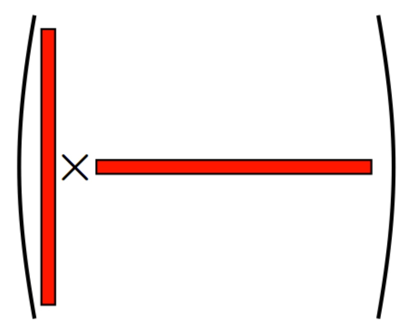
# Interpolatory low-rank approximations

An efficient method to construct low-rank approximation is to use interpolation formulas:

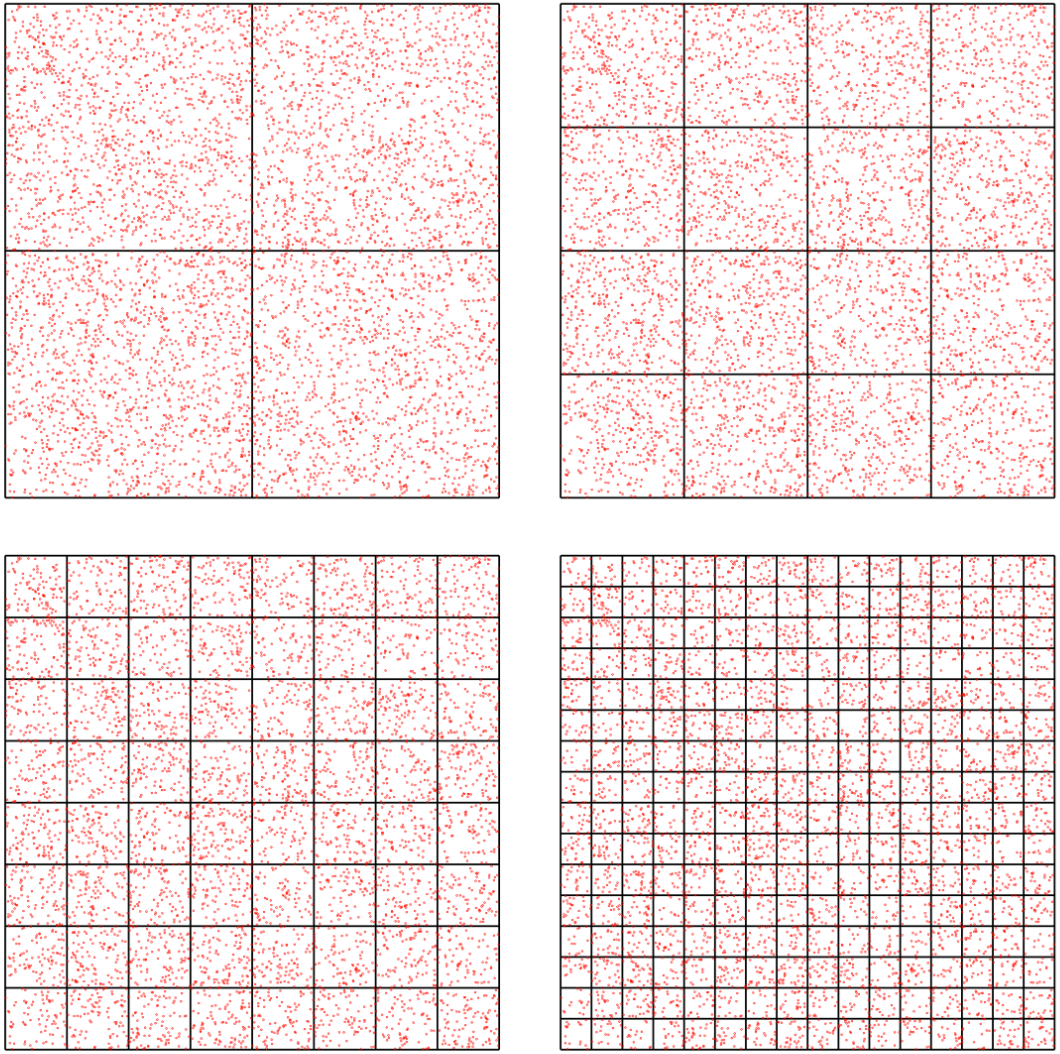
$$K(x,y) \approx \sum_{k=1}^{r} w_k(x) K(\bar{x}_k, y) \approx \sum_{k=1}^{r} w_k(x) \left( \sum_{j=1}^{r} w_j(y) K(\bar{x}_k, \bar{y}_j) \right)$$




$$K_{n\times n} = \underbrace{(T_{\text{inter}})_{n\times r} \tilde{K}_{r\times r} (T_{\text{anter}})_{r\times n}}_{\text{Low rank}} + \epsilon$$

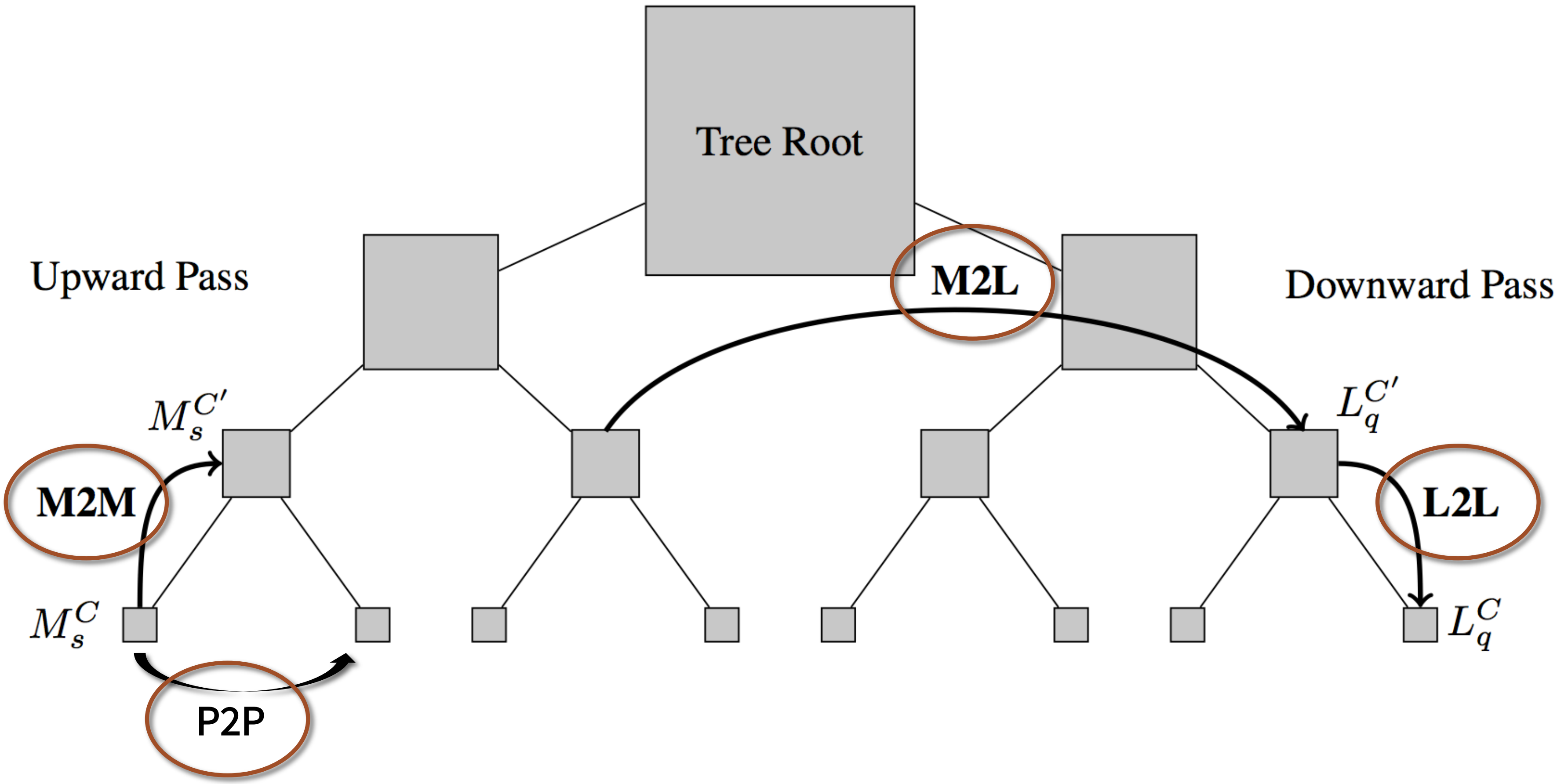


# FMM TREE

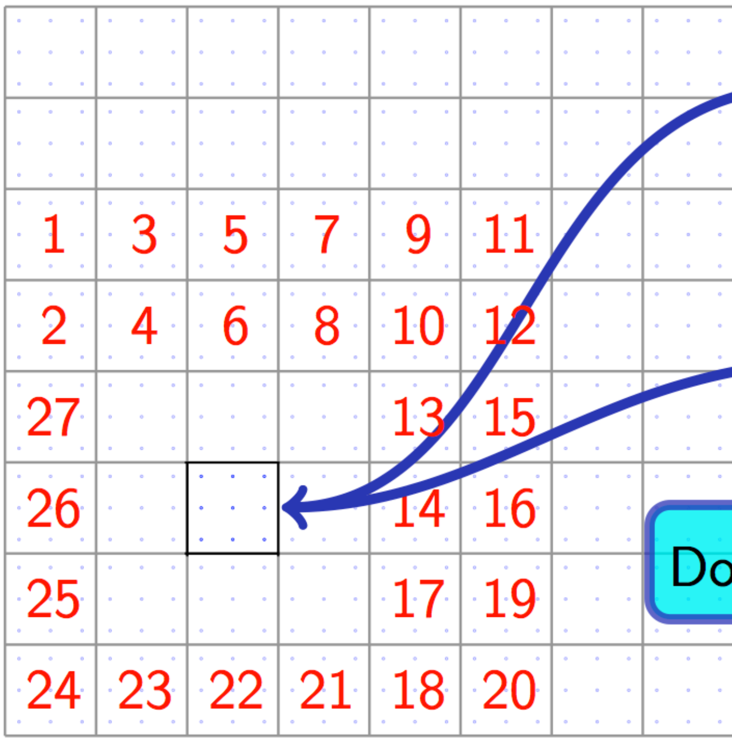


2D quad tree

## FMM OPERATORS



# M2L AND L2L




**L2L at level 3**

Interpolate from parent the far-field effect.

**M2L at level 3**

Compute the interaction from well-separated boxes.

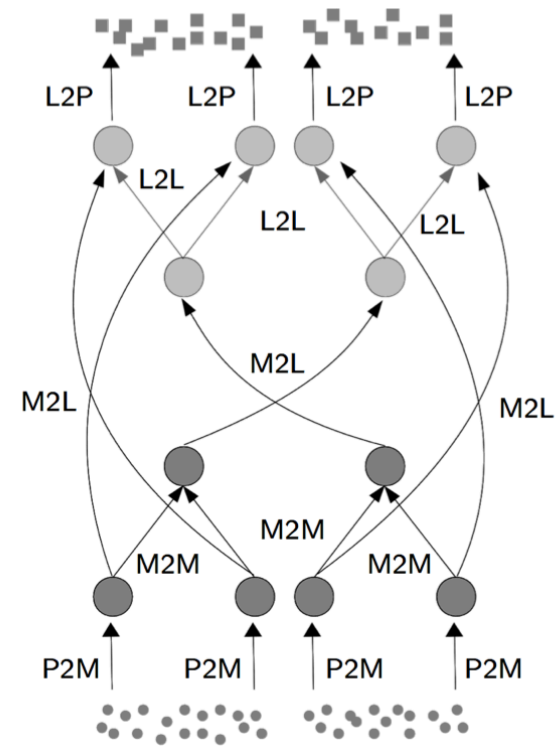
Do this for all boxes. Repeat at all levels.

The total computational cost for all these operations is $\mathcal{O}(rN)$

# PARALLEL IMPLEMENTATION

# PARALLELIZATION OF THE FMM

| Task | Concurrency | Load-balancing |
|---|---|---|
| M2L (lower level) | High | Homogeneous |
| M2L (higher level) | **Low** | Homogeneous |
| P2P | High | **Heterogeneous** |
| M2M/L2L | **Low** | Homogeneous |




Directed acyclic graph of tasks

# Classical OpenMP parallelization

```
function FMM(tree)
    // Near-field
    P2P(tree.levels[tree.height-1]);
    // Far-field
    P2M(tree.levels[tree.height-1]);
    forall the level l from tree.height-2 to 2 do
        M2M(tree.levels[l]);
    forall the level l from 2 to tree.height-2 do
        M2L(tree.levels[l]);
        L2L(tree.levels[l]);
    M2L(tree.levels[tree.height-1]);
    L2P(tree.levels[tree.height-1]);
```

The loop over cells at a given level are parallelized:

```
function M2L(level)
    #pragma omp parallel for
        foreach cell cl in level.cells do
            kernel.m2l(cl.local,
                       cl.far_field.multipole);
```

## IMPROVING THE SCALABILITY

- **When running on a large number of cores or if the computational problem is too small, one may run into performance issues, e.g.:**
  - some cores may become idle because there is no enough concurrent work or
  - many cores are done and only a few are still executing.
- **The top of the tree represents a parallel bottleneck: limited amount of work is available.**
- **The P2P tasks can be highly heterogeneous: long tasks are mixed up with short tasks.**
- **Solution:**
  - Mix parallel sections with more sequential ones.
  - Try to execute long tasks first and finish with short tasks.

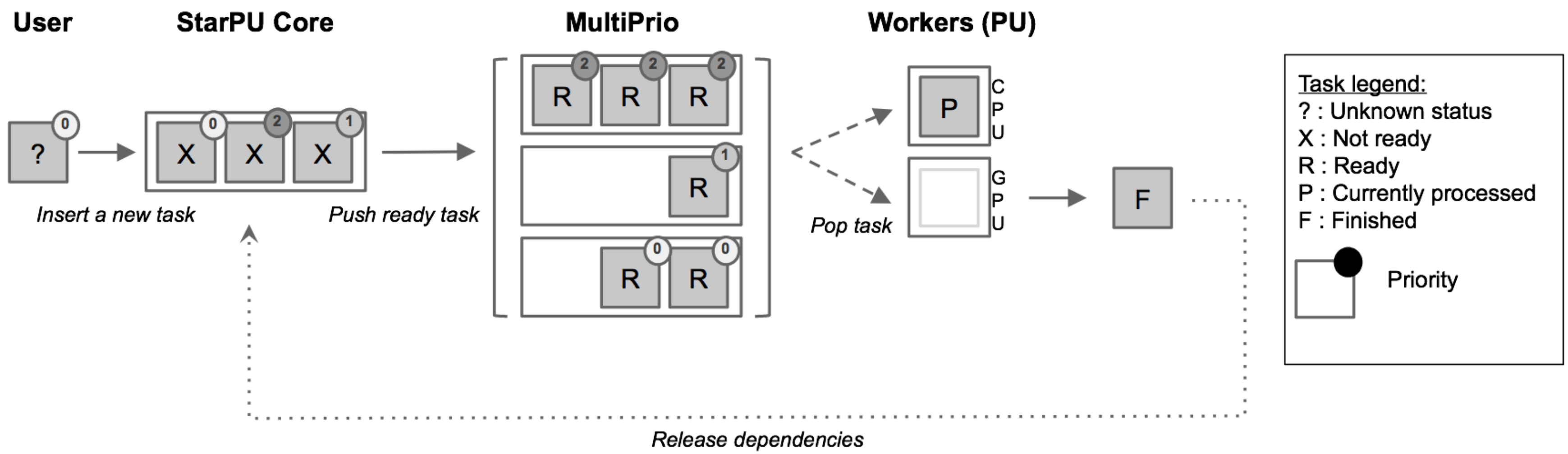
# Interleaving far field and near field calculations

```
#pragma omp parallel
    #pragma omp sections
        #pragma omp section
            P2Ptask(tree.levels[tree.height-1])
        #pragma omp section
            P2Mtask(tree.levels[tree.height-1])
            forall the level l from tree.height-2 to 2 do
                M2Mtask(tree.levels[l])
            forall the level l from 2 to tree.height-2 do
                M2Ltask(tree.levels[l])
                L2Ltask(tree.levels[l])
            M2Ltask(tree.levels[tree.height-1])
    #pragma omp single
        L2Ptask(tree.levels[tree.height-1])
```

```
function M2Ltask(level)
    foreach block of cells bl in level.blocks do
        #pragma omp task
            m2l(bl.local, bl.far_field.multipole);
    #pragma omp taskwait;
```

# BETTER YET: PTHREADS WITH RUNTIME SCHEDULER

Example of scheduling (implemented using Pthreads) based on priority lists:




**P2M > M2M > L2L(l) > P2P (large) > M2L(l) > P2P (small) > L2P**

Task priority rule

- Task insertion and retrieval.
- Use mutexes and condition variables in the implementation.