Sai

Dynamic Mapping (dynamic load balancing)

Centralized | Distributed

Centralized

→ All executable Tasks
- Common central Data Structure
- special process (Master + slaves)
- subset of processes

→ A process with no work takes portion of available work from the central data structure / Master

→ New tasks are added to Central data structure / reported to the Master

→ Easier to implement than distributed

→ Limited Scalability

e.g. Quick Sort each row of an $n \times n$ matrix

$O(n)$ (?) | $O(n \log n)$

⇒ Equal number of rows to each process - imbalance

⇒ Maintain Central pool of indices of rows yet to be sorted

Process when idle can pick an index from this Pool (self - scheduling)

→ Good load balancing but bottlenecks in accessing shared work queue if each task does not have large enough computation

∴ To ease the bottleneck, assign more than one Task at a time ⇒ Chunk Scheduling

But what if there is a load-imbalance due to large chunk sizes?

⇒ Reduce chunk size as the program progresses (linear, non-linear, ...)

Distributed

→ Set of executable Tasks are distributed among processes

→ Processes exchange tasks at run time to balance work

→ No bottleneck of Centralized Schemes

Its Critical parameters are:

→ Pairing of Sender/receiver processes

→ Initiation of work transfer - Sender or Receiver?

→ Amount of work transfer

→ Point of work transfer

To be effective in Message passing platforms, size of (computations) Tasks should be larger than size of data associated with it.

Reduce Process Interaction Overheads

→ Volume of data
→ Frequency of Interaction
→ Spatial & Temporal pattern of Interactions

Techniques to Reduce Interaction Overheads:

→ Promote use of recently fetched data (Temporal locality)

→ Minimize the Overall volume of shared data that needs to be accessed by Concurrent processes

$$\frac{2n^2}{p} \quad Vs \quad \frac{n^2}{\sqrt{p}} + n^2$$

$$(1-d) \qquad\qquad (2-d)$$

→ Locally Compute, store intermediate results, perform shared data access only to place final results

```
for(i = 0; i < N; i++)
    dp += A[i] * B[i];
```

```
for( i = id * N/p ; i < (id+1)N/p ; i++)
    ldp += A[i] * B[i];

dp += ldp;
```

→ Minimize frequency of interactions as each interaction involves relatively high Startup cost

- Restructure the algorithm such that shared data are accessed and used in large pieces

Amortizes startup cost over multiple accesses (akin to large cache lines for spatial locality)

Leads to fewer messages on Message Passing systems

e.g. Sparse Matrix Vector Multiplication

P0 needs b[3], b[5] & b[7], say

$b[3], b[5] \rightarrow P1$; $b[7] \rightarrow P2$

Combine interaction with P1 to get b[3] & b[5] instead of 2 separate interactions.

→ Minimize Contention

Why? Multiple Tasks accessing same resource concurrently

Only one of the operations can proceed at a time, others are queued, proceed sequentially

e.g. Multiple simultaneous transmission of data over the same interconnect link

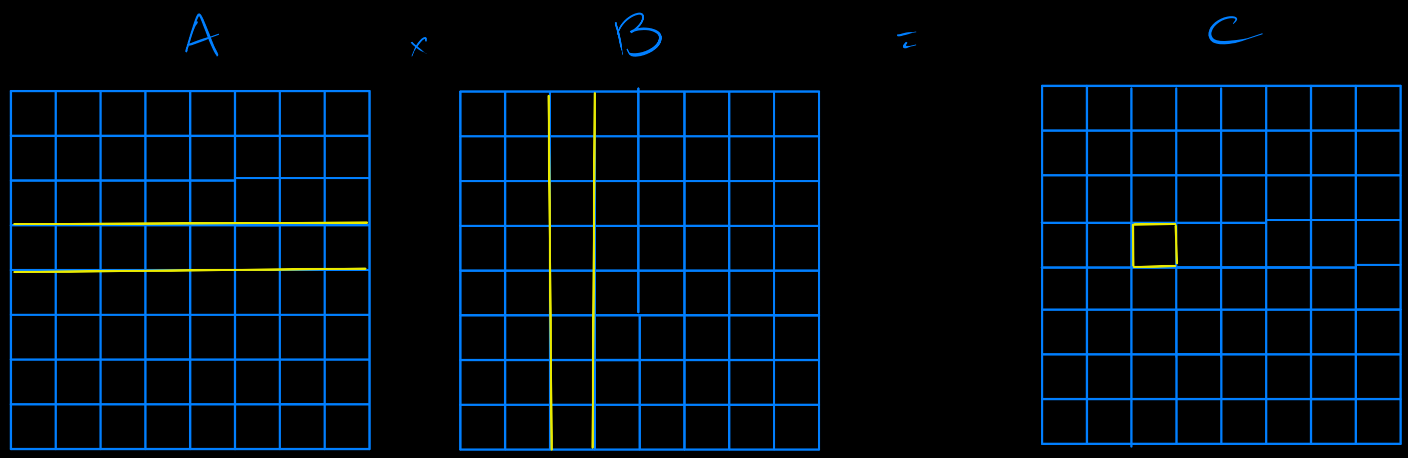
Multiple simultaneous access to same memory block

Multiple processes sending messages to the same process at the same time

$C = AB$ using 2-d partitioning, p tasks, p processes (1–1 mapping)

$\therefore$ Each Task to Compute $C_{i,j}$, $0 \le i, j < \sqrt{p}$

Let $p = 64$, $c_{ij} = \sum_{k=0}^{\sqrt{p}-1} A_{ik} * B_{kj}$



Processes Computing $c_{0,0}, c_{0,1}, \ldots, c_{0,\sqrt{p}-1}$ will attempt to read $A_{0,0}$ at the same time.

Processes Computing $c_{0,0}, c_{1,0}, \ldots, c_{\sqrt{p}-1,0}$ will attempt to read $B_{0,0}$ at the same time.

$\Rightarrow$ Contention in NUMA shared address space & Message passing platforms

$\therefore$ Redesign the algorithm to access data in Contention-free patterns.

$$c_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} * B_{(i+j+k)\%\sqrt{p},j}$$

Process Computing $c_{0,0}$ accesses : $A_{0,0}$

" " $c_{0,1}$ " : $A_{0,1}$

$\vdots$

$c_{0,7}$ " : $A_{0,7}$

Process computing $C_{5,0}$ accesses : $A_{5,5}$

$C_{5,1}$: $A_{5,6}$

$C_{5,2}$: $A_{5,7}$

$C_{5,3}$: $A_{5,0}$

$C_{5,4}$: $A_{5,1}$

$C_{5,5}$: $A_{5,2}$

$C_{5,6}$: $A_{5,3}$

$C_{5,7}$: $A_{5,4}$

[Centralized schemes in Dynamic Mapping are frequent source of contention for shared data structures $\Rightarrow$ choose distributed mapping]

→ Overlap Computations with Interactions

- Perform useful computations while waiting for shared data to arrive after initiating the interaction.
  - Initiate interaction early enough by identifying computations that can be performed before the interaction.
  - If one Task blocks on an interaction, and if there is another Task available to the same process, execute the other Task.

- Initiate advance work transfer if the process can anticipate running out of current work.

- Requires support from the programming paradigm and underlying hardware

  e.g. Non-Blocking Message Passing primitives

  Prefetching hardware / compiler to advance loads

→ Replicating Data or Computations

- For read-only shared data access, replicate it in each process, thereby avoiding interaction

- But it increases memory requirement linearly with number of concurrent processes.

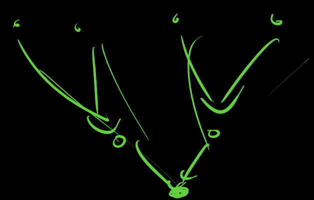
- Instead of sharing intermediate results, each process can compute them.

- Trade-off between interaction overhead to replicated computation

→ Optimized Collective Operations

- Static and regular pattern of interaction between a group of tasks e.g. Broadcast

  Reduce



- Used for → sharing data
→ Communication intensive computations
→ Synchronization

- Optimized implementation of collectives are available in libraries from parallel platform vendors
- In exceptional scenarios, develop one's own collective implementation.

→ Overlap Interactions with Other Interactions

- with hardware support, overlapping interactions between multiple pairs of processes can reduce effective volume of communication.

e.g. One to all broadcast from P_0 to P_1, P_2 & P_3

Naive algorithm:

$P_0 \xrightarrow[1]{} P_1 \xrightarrow[2]{} P_2 \xrightarrow[3]{} P_3$ = 3 units of time

Overlapped:

$P_0 \xrightarrow[1]{} P_1$ $\quad$ $P_0 \xrightarrow{} P_2$, $P_1 \xrightarrow[2]{} P_3$ = 2 units of time

What if 4 messages need to be sent?

$P_0 \xrightarrow{m_0} P_1$ $\quad$ $P_0 \xrightarrow{m_0} P_2$, $P_1 \xrightarrow{m_0} P_3$ $\qquad$ $P_0 \xrightarrow{m_1} P_1$ $\quad$ $P_0 \xrightarrow{m_1} P_2$, $P_1 \xrightarrow{m_1} P_3$

$P_0 \xrightarrow{m_2} P_1$ $\quad$ $P_0 \xrightarrow{m_2} P_2$, $P_1 \xrightarrow{m_2} P_3$ $\qquad$ $P_0 \xrightarrow{m_3} P_1$ $\quad$ $P_0 \xrightarrow{m_3} P_2$, $P_1 \xrightarrow{m_3} P_3$ = 8 units of time

If the naive algorithm is used :

$P_0 \xrightarrow{m_0} P_1$ $\quad$ $P_0 \xrightarrow{m_1} P_1$ $\quad$ $P_0 \xrightarrow{m_2} P_1$ $\quad$ $P_0 \xrightarrow{m_3} P_1$ $\quad$ $P_1 \xrightarrow{m_3} P_2$ $\quad$ $P_2 \xrightarrow{m_3} P_3$

$P_1 \xrightarrow{m_0} P_2$ $\quad$ $P_1 \xrightarrow{m_1} P_2$ $\quad$ $P_1 \xrightarrow{m_2} P_2$ $\quad$ $P_2 \xrightarrow{m_2} P_3$

$P_2 \xrightarrow{m_0} P_3$ $\quad$ $P_2 \xrightarrow{m_1} P_3$

= 6 units of time !

- An exceptional scenario for a user developed customized collective operation.