Segmented scan

Given a $n \times P$ matrix $A$, compute $n \times P$ matrix $B = S(A)$, where

$$B(i, j) = \sum_{k=1}^{j} A(i, k)$$

$$A_{\text{odd}} = [A(:, 1), A(:, 3), \ldots A(:, P - 1)],$$

$$A_{\text{even}} = [A(:, 2), A(:, 4), \ldots A(:, P)].$$

Now, observe that $B_{\text{even}} = S(A_{\text{odd}} + A_{\text{even}})$ and that $B_{\text{odd}} = B_{\text{even}} - A_{\text{even}}$.

The above version is a ‘postfix’ sum, a ‘prefix’ sum $B = R(A)$ is more standard

$$B(i, j) = \sum_{k=1}^{j-1} A(i, k)$$

Now, $B_{\text{even}} = R(A_{\text{odd}} + A_{\text{even}})$ and $B_{\text{odd}} = B_{\text{even}} + A_{\text{even}}$. Neither version requires an additive inverse. A scan is a prefix sum with an arbitrary $+$ operator.
Parallel segmented scan

The parallel prefix sum is the first parallel algorithm many people learn

\[ T_{\text{scan}}(P) = T_{\text{scan}}(P/2) + 2 = 2 \log_2(P) \]

for \( T \in \{\text{computation, communication, synchronization}\} \).
So we can trivially get

\[ T_{\text{seg-scan}}(n, P) = T_{\text{seg-scan}}(n, P/2) + 2 \cdot \alpha + 2n \cdot \beta = 2 \log_2(P) \cdot \alpha + 2n \log_2(P) \cdot \beta \]

MPI::Scan does the trivial algorithm :(

Note 1: the \( n \) scans are \textit{independent}
Note 2: parallel scan discards half the processors at each step

Butterfly Idea: assign \( n/2 \) of the scans to the other half of the processors

\[ T_{\text{seg-scan}}(n, P) = T_{\text{seg-scan}}(n/2, P/2) + 2 \cdot \alpha + (n/2) \cdot \beta = 2 \log_2(P) \cdot \alpha + n \cdot \beta \]

BSP Idea: transpose \( A \) and have each processor compute \( n/P \) scans sequentially
Senders vs receivers in a wrapped butterfly

We proved in lecture that the senders in the wrapped butterfly (Träff and Ripke) algorithm are independent

- I thought the showing this for receivers would require some work
- some students were more clever than me...
- the set of receivers at the next level is the set of senders in the previous with a flipped bit
- if $x \neq y$, flipping the same bit preserves the inequality
  - if we flip a bit that is different in $x$ and $y$, the bits remain different
- HW 1 take-away: *simplicity is attained by finding the right perspective*
Homework 2

- problem 1 is Strassen’s algorithm
  - recursion dragon is back
  - algorithms are given, your task: analysis
  - should be analogous to recursive MM and LU
- problem 2 is radix sort
  - algorithm given, last part requires minor modification
  - your primary task is again cost analysis
  - uses HW 1 problem 1!
- if you did not complete HW 1, remember the lowest homework grade is disregarded, but not the second lowest...
Recursive LU factorization: analysis

LU requires two recursive calls and $O(1)$ matrix multiplications

$$T_{LU}(n, P) = 2T_{LU}(n/2, P) + O\left(\log(P) \cdot \alpha + \frac{n^2}{P^{2/3}} \cdot \beta\right)$$

the bandwidth cost decreases geometrically (by a factor of 2) at each level.
If we allgather the matrix at the base cases, each has a cost of

$$T_{LU}(n_0, P) = O(\log(P) \cdot \alpha + n_0^2 \cdot \beta)$$

Q: What choice of $n_0$ makes the base cases have bandwidth cost less than $\frac{n^2}{P^{2/3}}$?

$$T_{bc}(n, n_0, P) = \frac{n}{n_0} T_{LU}(n_0, P)$$

A: we would want select is $n_0 = n/P^{2/3}$, giving a total cost of

$$T_{LU}(n, P) = O(P^{2/3} \cdot \log(P) \cdot \alpha + \frac{n^2}{P^{2/3}} \cdot \beta)$$

In the BSP model, we lose the $\log(P)$ factors in synchronization cost.
Recursive triangular inversion: analysis

The two recursive calls within triangular inversion are independent, so we can perform them simultaneously with half of the processors

\[ T_{\text{Tri-Inv}}(n, P) = T_{\text{Tri-Inv}}(n/2, P/2) + O(T_{\text{MM}}(n, P)) \]

\[ = T_{\text{Tri-Inv}}(n/2, P/2) + O\left(\log(P) \cdot \alpha + \frac{n^2}{P^{2/3}} \cdot \beta\right) \]

with base-case cost (sequential execution)

\[ T_{\text{Tri-Inv}}(n_0, P) = O(\log(P) \cdot \alpha + n_0^2 \cdot \beta) \]

the bandwidth cost goes down at each level and we can execute the base-case sequentially when \( n_0 = n/P^{1/3} \), with a total cost of

\[ T_{\text{Tri-Inv}}(n, P) = O\left(\log(P)^2 \cdot \alpha + \frac{n^2}{P^{2/3}} \cdot \beta\right) \]

So triangular inversion has \textit{logarithmic depth} while LU has \textit{polynomial depth}, but using inversion within LU naively would raise the LU latency by another log factor
Memory-efficient recursive LU factorization

In the analysis of recursive LU, we assumed

\[ T_{MM}(n, P) = O\left(\log(P) \cdot \alpha + n^2 / P^{2/3} \cdot \beta\right) \]

which requires \( n^2 / P^{2/3} \) memory, \( P^{1/3} \) more than minimal

What if we have only \( cn^2 / P \) memory for some \( c \in [1, P^{1/3}] \)?

\[ T_{MM}(n, P, c) = O\left(\sqrt{P/c^3} \log(P) \cdot \alpha + n^2 / \sqrt{cP} \cdot \beta\right) \]

Q: Does the additional MM latency cost raise the LU latency cost?

A/Q: Naively yes, but could we do something about it?

A: Yes, we could increase \( c \) for small subproblems.

What should we set the base case dimension to (previously \( n_0 = n/P^{2/3} \))?

\[ T_{bc}(n, n_0) = O\left((n/n_0)(\log(P) \cdot \alpha + n_0^2 \cdot \beta)\right) \]

\[ T_{bc}\left(n, \frac{n}{\sqrt{cP}}\right) = O\left(\sqrt{cP}\left(\log(P) \cdot \alpha + \frac{n^2}{cP} \cdot \beta\right)\right) = O\left(\sqrt{cP} \log(P) \cdot \alpha + \frac{n^2}{\sqrt{cP}} \cdot \beta\right) \]
Short pause
Course projects and homework

Course projects

- the choice of project will be flexible
- doing something in your current research area is encouraged
- first proposal deadline pushed back a week to Sep 28
- I am happy to give feedback or ideas over email or in person

Homework 2

- is due Sep 21
- post questions on Piazza or come to office hours!
2.5D LU factorization
2.5D LU factorization
Memory-efficient LU factorization

2.5D LU factorization

(A)  

(B)  

(C)  

(D)