Design of Parallel and High-Performance Computing:
Distributed-Memory Models and Algorithms

Edgar Solomonik

ETH Zürich

December 9, 2014

Summary

Lecture overview

- Review: α-β communication cost model
- LogP model: short message broadcast
- LogGP model: handling long messages
- Algorithms for long-message broadcasts
- Other types of collective communication
- Bulk Synchronous Parallel (BSP) model
- PGAS languages / one-sided communication
- Communication-avoiding algorithms
- Overview and final comments

A simple model for point-to-point messages

The cost of sending a message of size s bytes from one processor to any other is

$$\alpha + s \cdot \beta$$

The latency (per message) cost is α, while the inverse bandwidth (per byte) cost is β

What is the cost if the ith processor sends a message of size s to the $(i+1)$th processor for $i \in [1, p]$?

- it depends on whether the messages are dependent!
- the 'total work' is $p \cdot (\alpha + s \cdot \beta)$
- the parallel execution (critical path) cost if the messages are sent simultaneously is only $\alpha + s \cdot \beta$
- the parallel execution (critical path) cost if the messages are sent in sequence is $p \cdot (\alpha + s \cdot \beta)$

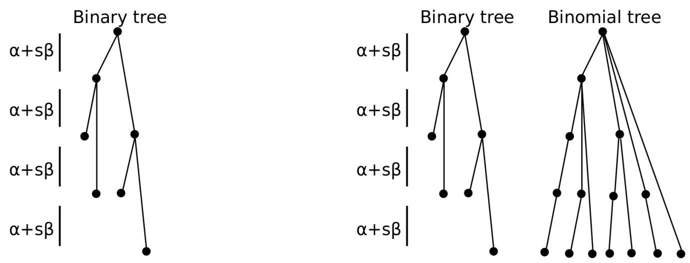
Small message broadcasts in the α-β model

The cost of a **binary** tree broadcast of a message of size s is

$$2(\log_2(p+1) - 1) \cdot (\alpha + s \cdot \beta)$$

The cost of a **binomial** tree broadcast of a message of size s is

$$\log_2(p+1) \cdot (\alpha + s \cdot \beta)$$



The LogP model

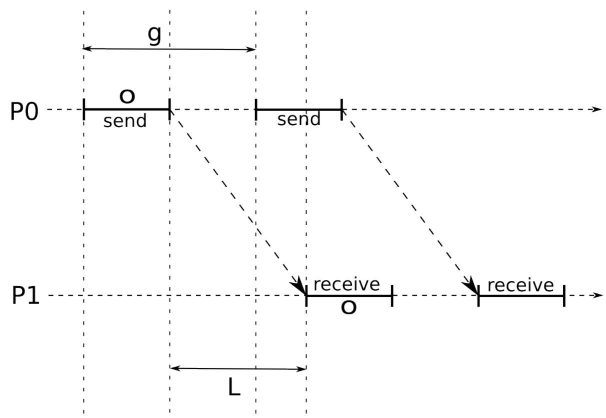
The α-β model is simplistic in its representation of a point-to-point message

- assumes both sender and receiver block until completion
- precludes overlap between multiple messages or between computation and communication

The LogP model (Culler et al. 1996) enables modelling of overlap for the transfer of a fixed-size message

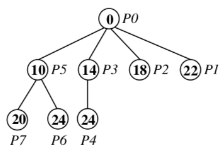
- L – network **latency** cost (processor not occupied)
- o – sender/receiver sequential messaging **overhead** (processor occupied)
- g – inverse injection rate, i.e. **gap** between messages (processor not occupied)
- P – number of **processors**

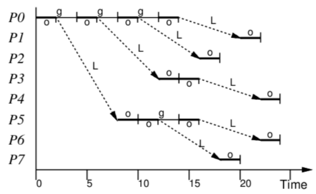
Messaging in the LogP model



Small-message broadcasts in the LogP model

Same idea as binomial tree, forward message as soon as it is received, keep forwarding until all nodes obtain it (Karp et al. 1993)





The LogGP model

The parameter g in the LogP model is associated with an implicit packet size

- this injection rate implies a fixed-sized packet can be sent anywhere after a time interval of g
- modern computer networks do not have a small fixed packet size and achieve higher bandwidth for large messages

The LogGP model (Alexandrov et al. 1997) introduces another bandwidth parameter G, which dictates the bandwidth achieved by large point-to-point messages

- g – **gap**; minimum time interval between consecutive message transmissions
- G – **Gap per byte**; time per byte for a long message

The LogGP model

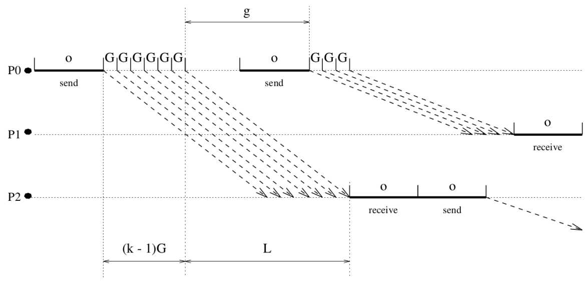


Diagram taken from: Alexandrov, A., Ionescu, M. F., Schauser, K. E., and Scheiman, C. LogGP: incorporating long messages into the LogP model–one step closer towards a realistic model for parallel computation. ACM SPAA, July 1995.

Large-message broadcasts

Lets now consider broadcasts of a message of a large size s (we will assume $s \geq P$)

- it is inefficient to send the entire message at once
- pipelining the message is critical (recursive halving is good too)
- the goal is to get as close to (in the α-β model sense) have a bandwidth cost of $s \cdot \beta$
- binomial tree is very bad, root sends message $\log(p)$, i.e. the cost is approximately

$$\log(p) \cdot s \cdot \beta$$

Pipelined binary tree broadcast

Send a fixed-size packet to left child then to right child (entire message of size s)

- the LogP model has a fixed packet size k_{LogP} associated with g and yields the cost

$$T_{\text{PBT}}^{\text{LogP}} \approx \log(P) \cdot (L + 3\max(o, g)) + 2(s/k_{\text{LogP}}) \cdot \max(o, g)$$

- in the LogGP model we can select a packet size k and obtain the cost

$$T_{\text{PBT}}^{\text{LogGP}}(k) \approx \log(P) \cdot (L + 3\max(o, g) + 2G \cdot k) + 2(s/k) \cdot (\max(o, g) + G \cdot k)$$

minimizing the packet size k, $T_{\text{PBT}}^{\text{LogGP}}(k_{\text{opt}}^{\text{LogGP}}) = \min_k T_{\text{PBT}}^{\text{LogGP}}(k)$ (via e.g. differentiation by k) we obtain the optimal packet size

$$k_{\text{opt}}^{\text{LogGP}} = \sqrt{s/\log(P)} \cdot \sqrt{\frac{\max(o, g)}{G}}$$

so the best packet size, depends not only on architectural parameters, but also on dynamic parameters: the number of processors and message size

Pipelined binary tree broadcast contd.

In LogP we obtained

$$T_{\text{PBT}}^{\text{LogP}} \approx \log(P) \cdot (L + 3\max(o, g)) + 2(s/k_{\text{LogP}}) \cdot \max(o, g)$$

In LogGP we obtained,

$$T_{\text{PBT}}^{\text{LogGP}}(k) \approx \log(P) \cdot (L + 3\max(o, g) + 2G \cdot k) + 2(s/k) \cdot (\max(o, g) + G \cdot k)$$

$$k_{\text{opt}}^{\text{LogGP}} = \sqrt{s/\log(P)} \cdot \sqrt{\frac{\max(o, g)}{G}}$$

- in the α-β model for a packet size of k, we obtain the cost

$$T_{\text{PBT}}^{\alpha,\beta}(k) \approx 2(\log(P) + s/k)(\alpha + k \cdot \beta)$$

with a minimal packet size of

$$k_{\text{opt}}^{\alpha,\beta} = \sqrt{s/\log(P)} \cdot \sqrt{\frac{\alpha}{\beta}}$$

Pipelined binary tree broadcast conclusions

The LogP model is inflexible, while the LogGP and the α-β models capture the key input and architectural scaling dependence

$$T_{\mathrm{PBT}}^{\alpha,\beta}(k) \approx 2(\log(P) + s/k)(\alpha + k\cdot\beta)$$

$$k_{\mathrm{opt}}^{\alpha,\beta} = \sqrt{s/\log(P)} \cdot \sqrt{\frac{\alpha}{\beta}}$$

The minimized cost in the α-β model is

$$T_{\mathrm{PBT}}^{\alpha,\beta}(k_{\mathrm{opt}}^{\alpha,\beta}) \approx 2\left(\log(P) + \sqrt{s\cdot\log(P)}\cdot\sqrt{\frac{\beta}{\alpha}}\right)\cdot\left(\alpha + \sqrt{\frac{s}{\log(P)}}\cdot\sqrt{\alpha\cdot\beta}\right)$$
$$\approx 2\log(P)\cdot\alpha + \sqrt{2s\cdot\log(P)}\cdot\sqrt{\alpha\cdot\beta} + 2s\cdot\beta$$

Q: Could we get rid of the factor of two constant in the $O(s\cdot\beta)$ cost?
A: Not so long as the root sends two copies of the whole message...

Double Tree (not the Hilton hotel chain, but still fancy)

Note that the leaves of a binary tree, $(P-1)/2$ processors, send nothing, while the internal nodes do all the work

Idea of Double Pipelined Binary Tree: use two binary trees, where every non-root processor is a leaf in one and an internal node in the other, send half of the message down each tree

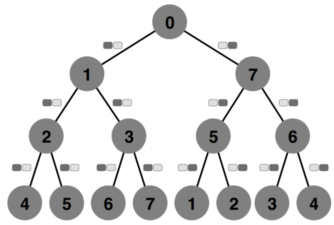


Diagram taken from: Hoefler, Torsten, and Dmitry Moor. "Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations."

Double pipelined binary tree

The cost of the double pipelined binary tree is essentially the same as the cost of a single pipelined binary tree with half the message size roughly half the cost of a single pipelined binary tree

$$T_{\mathrm{DPBT}} \approx 2\log(P)\cdot\alpha + \sqrt{s\cdot\log(P)}\cdot\sqrt{\alpha\cdot\beta} + s\cdot\beta$$

for a sufficiently large message size (s) this is twice as fast as a single pipelined binary tree

Other types of collective communication

We can classify collectives into four categories

- **One-to-All**: Broadcast, Scatter
- **All-to-One**: Reduce, Gather
- **All-to-One + One-to-All**: Allreduce (Reduce+Broadcast), Allgather (Gather+Broadcast), Reduce-Scatter (Reduce+Scatter), Scan
- **All-to-All**: All-to-all

MPI (Message-Passing Interface) provides all of these as well as variable size versions (e.g. (All)Gatherv, All-to-allv), see online for specification of each routine

Tree collectives

We have demonstrated how (double/pipelined) binary trees and binomial trees can be used for broadcasts

- *A reduction may be done via any broadcast tree with the same communication cost, with reverse data flow*

$$T_{\mathrm{reduce}} = T_{\mathrm{broadcast}} + \text{cost of local reduction work}$$

Scatter is strictly easier than broadcast, pipeline half message to each child in a binary tree

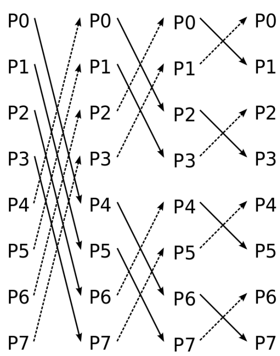
$$T_{\mathrm{scatter}} \approx 2\log(P)\cdot\alpha + s\cdot\beta$$

- *A gather may be done via the reverse of any scatter algorithm*:

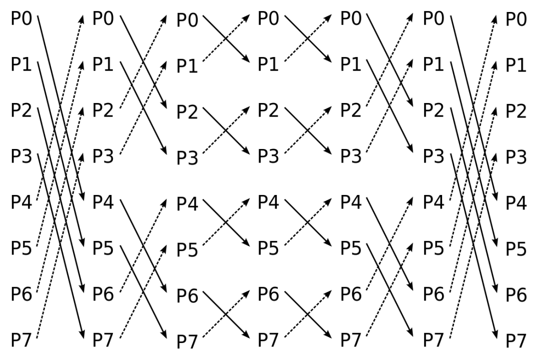
$$T_{\mathrm{gather}} = T_{\mathrm{scatter}}$$

All-to-One + One-to-All collectives can be done via two trees, but is this most efficient? What about **All-to-All** collectives?

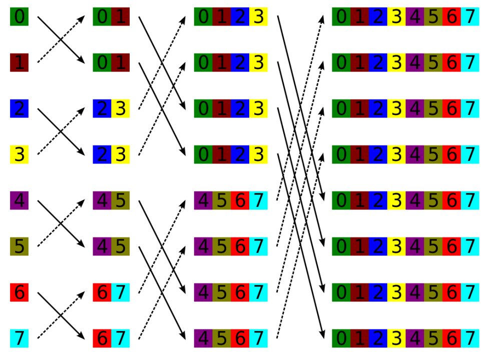
Butterfly network



Butterfly network



Butterfly Allgather (recursive doubling)



Cost of butterfly Allgather

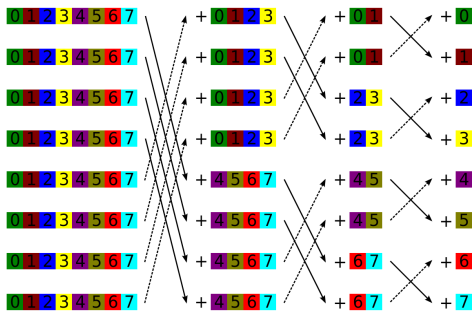
The butterfly has $\log(p)$ levels. The size of the message doubles at each level until all s elements are gathered, so the total cost is

$$T_{\text{allgather}} = \alpha \cdot \log(p) + \beta \cdot \sum_{i=1}^{\log(p)} s/2^i = \alpha \cdot \log(p) + \beta \cdot s$$

The geometric summation in the cost is typical of butterfly algorithms and critical to their efficiency

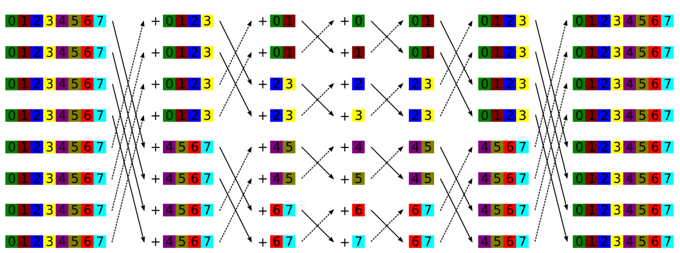
- no pipelining necessary to get rid of $\log(p)$ factor on bandwidth cost!

Butterfly Reduce-Scatter (recursive halving)

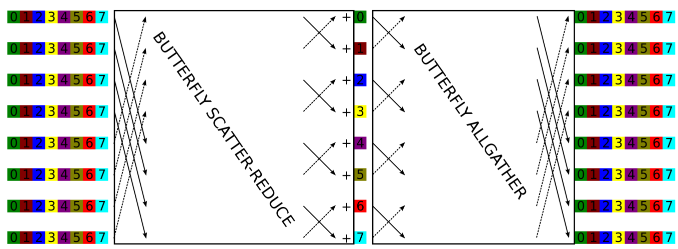


$$T_{\text{reduce-scatter}} = T_{\text{allgather}} + \text{cost of local reduction work}$$

Butterfly Allreduce

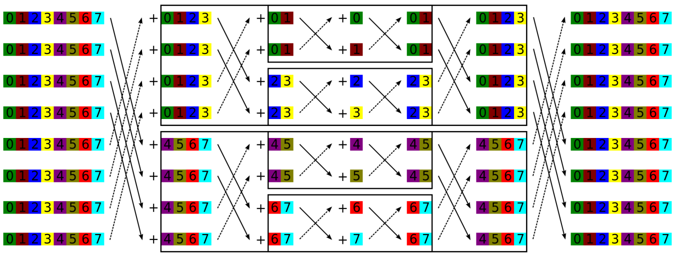


Butterfly Allreduce



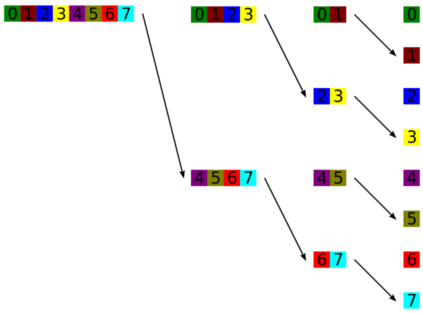
$$T_{\text{allreduce}} = T_{\text{reduce-scatter}} + T_{\text{allgather}}$$

Butterfly Allreduce: note recursive structure of butterfly



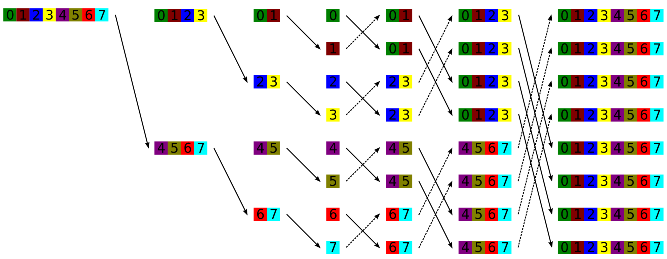
Its possible to do Scan (each processor ends up with a unique value of a prefix sum rather than the full sum) in a similar fashion, but also with operator application done additionally during recursive doubling (Allgather)

Butterfly Scatter

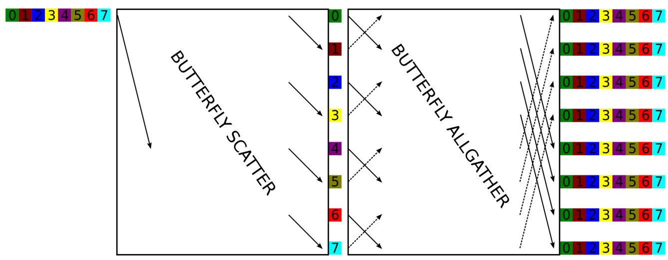


Question: Which tree is this equivalent to?

Butterfly Broadcast

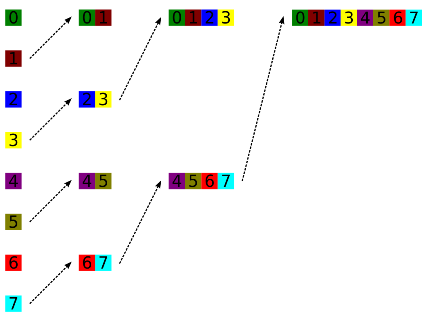


Butterfly Broadcast



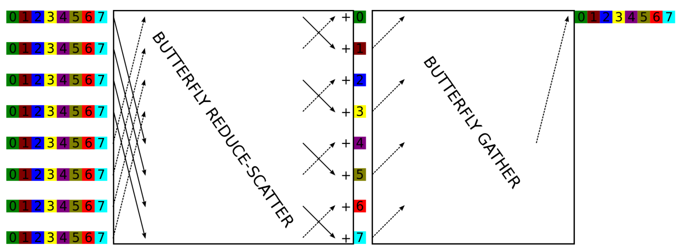
$$T_{\text{broadcast}} = T_{\text{scatter}} + T_{\text{allgather}}$$

Butterfly Gather



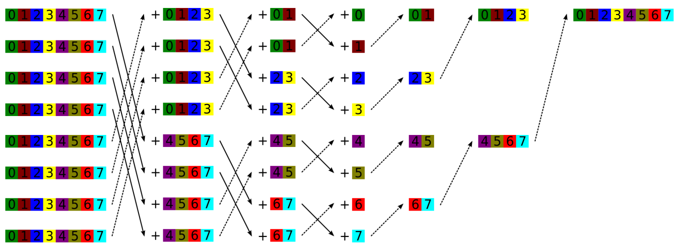
Question: Which other butterfly collective can utilize Gather as a subroutine?

Butterfly Reduce



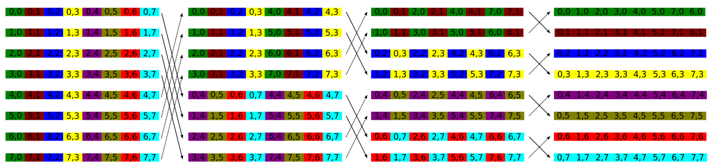
$$T_{\text{reduce}} = T_{\text{reduce-scatter}} + T_{\text{gather}}$$

Butterfly Reduce



$$T_{\text{reduce}} = T_{\text{reduce-scatter}} + T_{\text{gather}}$$

Butterfly All-to-All



Note that the size of the message stays the same at each level

$$T_{\text{all-to-all}} = \alpha \cdot \log(p) + \beta \cdot \sum_{i=1}^{\log(p)} s = \alpha \cdot \log(p) + \beta \cdot s \cdot \log(p)$$

Its possible to do All-to-All in less bandwidth cost (as low as $\beta \cdot s$ by sending directly to targets) at the cost of more message (as high as $\alpha \cdot p$ if sending directly)

BSP model definition

The Bulk Synchronous Parallel (BSP) model (Valiant 1990) is a theoretical execution/cost model for parallel algorithms

- execution is subdivided into supersteps, each associated with a global synchronization
- within each superstep each processor can send and receive up to h messages (called an h-relation)
- the cost of sending or receiving h messages of size m is $h \cdot m \cdot \hat{g}$
- the total cost of a superstep is the max over all processors at that superstep
- when $h = 1$ the BSP model is closely related to the α-β model with $\beta = \hat{g}$ and LogGP mode with $G = \hat{g}$
- we will focus on a variant of BSP with $h = p$ and for consistency refer to $\hat{g}$ as β and the cost of a synchronization as α

Synchronization vs latency

By picking $h = p$, we allow a global barrier to execute in the same time as the point-to-point latency

- this abstraction is good if the algorithm's performance is not expected to be latency-sensitive
- messages become non-blocking, but progress must be guaranteed by barrier
- collectives can be done in linear bandwidth cost with $O(1)$ supersteps
- enables high-level algorithm development: how many collective protocols does the algorithm need to execute?
- global barrier may be a barrier of a subset of processors, if BSP is used recursively

Nonblocking communication

The paradigm of sending non-blocking messages then synchronizing later is sensible

- MPI provides non-blocking 'I(send/recv)' primitives that may be 'Wait'ed on in bulk (these are slightly slower than blocking primitives, due to buffering)
- MPI and other communication frameworks also provide one-sided messaging primitives which are zero-copy (no buffering) and very efficient
- one-sided communication progress must be guaranteed by a barrier on all or a subset of processors (or MPI Win Flush between a pair)

(Reduce-)Scatter and (All)Gather in BSP

When $h = p$ all discussed collectives that require a single butterfly can be done in time $T_{\text{butterfly}} = \alpha + s \cdot \beta$ i.e. they can all be done in one superstep

- Scatter: root sends each message to its target (root incurs $s \cdot \beta$ send bandwidth)
- Reduce-Scatter: each processor sends its portion to every other processor (every processor incurs $s \cdot \beta$ send bandwidth)
- Gather: send each message to root (root incurs $s \cdot \beta$ receive bandwidth)
- Allgather: each processor sends its portion to every other processor (every processor incurs $s \cdot \beta$ send and receive bandwidth)

when $h < p$, we could perform the above algorithms using a butterfly with 'radix'$=h$ (number of neighbors at each butterfly level) in time $T_{\text{butterfly}} = \log_h(p) \cdot \alpha + s \cdot \beta$

Other collectives in BSP

The Broadcast, Reduce, and Allreduce collectives may be done as combinations of collectives in the same way as with Butterfly algorithms, using two supersteps

- Broadcast done by Scatter then Allgather
- Reduce done by Reduce-Scatter then Gather
- Allreduce done by Reduce-Scatter then Allgather

Sign of a good model: simpler, yet preserves qualitative results

However, (bad sign) BSP with $h = p$ can do all-to-all in $O(s)$ bandwidth and $O(1)$ supersteps (as cheap as other collectives), when $h < p$, the logarithmic factor on the bandwidth is recovered

Systems for one-sided communication

BSP employs the concept of non-blocking communication, which presents practical challenges

- to avoid buffering or additional latency overhead, the communicating processor must know be aware of the desired buffer location of the remote processor
- if the location of the remote buffer is known, the communication is called 'one-sided'
- with network hardware known as Remote Direct Memory Access (RDMA) one-sided communication can be accomplished without disturbing the work of the remote processor

One-sided communication transfers are commonly be formulated as

- 'Put' – send a message to a remote buffer
- 'Get' – receive a message from a remote buffer

Partitioned Global Address Space (PGAS)

PGAS programming models facilitate non-blocking remote memory access

- they allow declaration of buffers in a globally-addressable space, which other processors can access remotely
- Unified Parallel C (UPC) is a compiler-based PGAS language that allows direct indexing into globally-distributed arrays (Carlson et al 1999)
- Global Arrays (Nieplocha et al 1994) is a library that supports a global address space via a one-sided communication layer (e.g. ARMCI, Nieplocha et al 1999)
- MPI supports one-sided communication via declaration of 'windows' that declare remotely-accessible buffers

Matrix multiplication

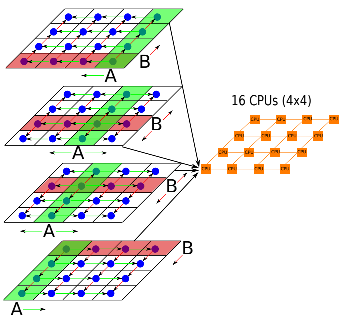
Matrix multiplication of n-by-n matrices A and B into C, $C = A \cdot B$ is defined as, for all i, j,

$$C[i,j] = \sum_k A[i,k] \cdot B[k,j]$$

A standard approach to parallelization of matrix multiplication is commonly referred to as SUMMA (Agarwal et al 1995, Van De Geijn et al 1997), which uses a 2D processor grid, so blocks A_{lm}, B_{lm}, and C_{lm} are owned by processor $P[l,m]$

- SUMMA variant 1: iterate for $k = 1$ to $\sqrt{p}$ and for all $i, j \in [1, \sqrt{p}]$
  - broadcast A_{ik} to $P[i,:]$
  - broadcast B_{kj} to $P[:,j]$
  - compute $C_{ij} = C_{ij} + A_{ik} \cdot B_{kj}$ with processor $P[i,j]$

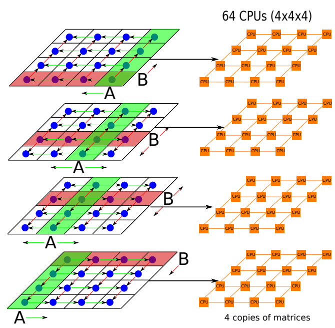
SUMMA algorithm



$$T_{\text{SUMMA}} = \sqrt{p} \cdot \left(2\log(p) \cdot \alpha + \frac{2n^2}{p} \cdot \beta\right) = 2\sqrt{p} \cdot \log(p) \cdot \alpha + \frac{2n^2}{\sqrt{p}} \cdot \beta$$

3D Matrix multiplication algorithm

Reference: Agarwal et al 1995 and others



$$T_{\text{3D-MM}} = 3\log(p) \cdot \alpha + \frac{3n^2}{p^{2/3}} \cdot \beta$$

LU factorization

The LU factorization algorithm provides a stable (when combined with pivoting) replacement for computing the inverse of a n-by-n matrix A,

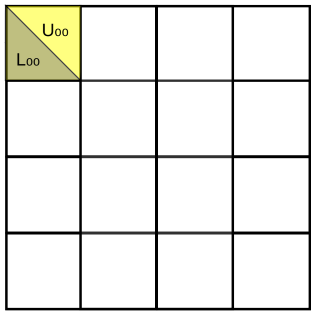
$$A = L \cdot U$$

where L is lower-triangular and U is upper-triangular is computed via Gaussian elimination, for $k = 1$ to n,

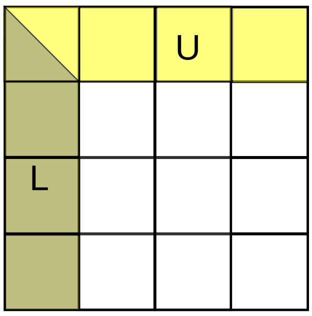
- set $L[k, k] = 1$ and $U[k, k : n] = A[k, k : n]$
- divide $L[k+1 : n, k] = A[k+1 : n, k]/U[k, k]$
- update Schur complement $A[k+1 : n, k+1 : n] = A[k+1 : n, k+1 : n] - L[k+1 : n, k] \cdot U[k, k+1 : n]$

this algorithm can be blocked analogously to matrix multiplication

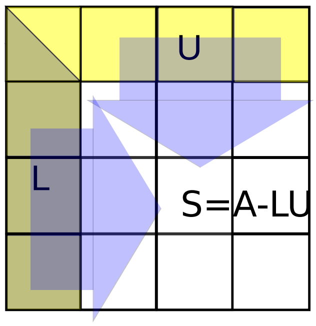
Blocked LU factorization



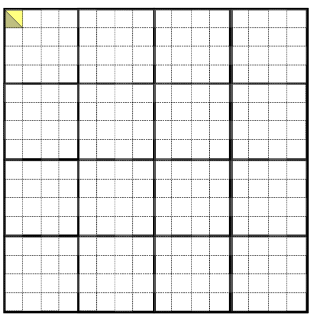
Blocked LU factorization



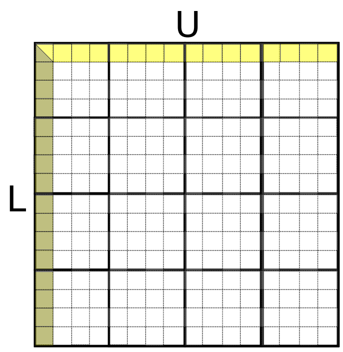
Blocked LU factorization



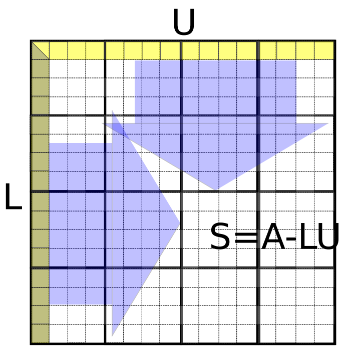
Block-cyclic LU factorization



Block-cyclic LU factorization



Block-cyclic LU factorization



Recursive matrix multiplication

Now lets consider a recursive parallel algorithm for matrix multiplication

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{22} &= A_{12} \cdot B_{21} + A_{22} \cdot B_{22} \end{aligned}$$

This requires 8 recursive calls to matrix multiplication of $n/2$-by-$n/2$ matrices, as well as matrix additions at each level, which can be done in linear time

Recursive matrix multiplication: analysis

If we execute all 8 recursive multiplies in parallel with $p/8$ processors, we obtain a cost recurrence of

$$T_{\mathrm{MM}}(n,p) = T_{\mathrm{MM}}(n/2, p/8) + O\left(\frac{n^2}{p} \cdot \beta\right) + O(\alpha)$$

The bandwidth cost is dominated by the base cases, where it is proportionate to

$$\left(n/2^{\log_8(p)}\right)^2 = (n/p^{\log_8(2)})^2 = (n/p^{1/3})^2 = n^2/p^{2/3}$$

for a total that we have seen before (3D algorithm)

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

Recursive LU factorization

LU factorization has the form

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & L_{12} \\ 0 & L_{22} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & 0 \\ U_{21} & U_{22} \end{bmatrix}$$

and can be computed recursively via

$$\begin{aligned} [L_{11}, U_{11}] &= \mathrm{LU}(A_{11}) \\ L_{21} &= A_{21} \cdot U_{11}^{-1} \\ U_{12} &= L_{11}^{-1} \cdot A_{12} \\ [L_{22}, U_{22}] &= \mathrm{LU}(A_{22} - L_{21} \cdot U_{12}) \end{aligned}$$

The inverses L_{11}^{-1} and U_{11}^{-1} may be obtained as part of the recursion in the first step (see Tiskin 2002 for details). There are two recursive calls to LU and 3 matrix multiplications needed at each step

Recursive LU factorization: analysis

The two recursive calls within LU factorization must be done in sequence, so we perform them with all the processors. We have to also pay for the cost of matrix multiplication at each level

$$\begin{aligned} T_{\mathrm{LU}}(n,p) &= 2T_{\mathrm{LU}}(n/2,p) + O(T_{\mathrm{MM}}(n,p)) \\ &= 2T_{\mathrm{LU}}(n/2,p) + O\left(\frac{n^2}{p^{2/3}} \cdot \beta + \log(p) \cdot \alpha\right) \end{aligned}$$

with base-case cost (sequential execution)

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

the bandwidth cost goes down at each level and we can execute the base-case sequentially when $n_0 = n/p^{2/3}$, with a total cost of

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

Conclusion and summary

Summary:

- important parallel communication models: α-β, LogP, LogGP, BSP
- collective communication: binomial trees are good for small-messages, pipelining and/or butterfly needed for large-messages
- collective protocols provide good building blocks for parallel algorithms
- recursion is a thematic approach in communication-efficient algorithms

Next semester:

- Consider taking "Research Topics in Software Engineering" next semester, it will have nothing to do with software engineering and lots to do with theory of parallel computation!

Backup slides