Heterogeneous solutions

- Graphics and Game processors
  - *Graphics Processing Units (GPUs), e.g., NVIDIA and ATI/AMD*
  - *Game processors, e.g., Cell for PS3*
  - *Parallel processor attached to main processor (APU)*
  - *Originally special purpose, getting more general*
  - *Programming model not yet mature*
- FPGAs – Field Programmable Gate Arrays
  - *Inefficient use of chip area*
  - *More efficient than multicore for some domains*
  - *Programming challenge now includes hardware design, e.g., layout*
- Tensor processing or other NeuralNet' units for machine learning

Towards parallel computing

- What are suitable performance metrics?
- What is the appropriate parallelism granularity (~ task size)?
- How to benefit from data locality?
- How to map, coordinate, and synchronize tasks?
- How to scheduling and load balance applications?

Parallel programming is much more difficult than sequential programming.

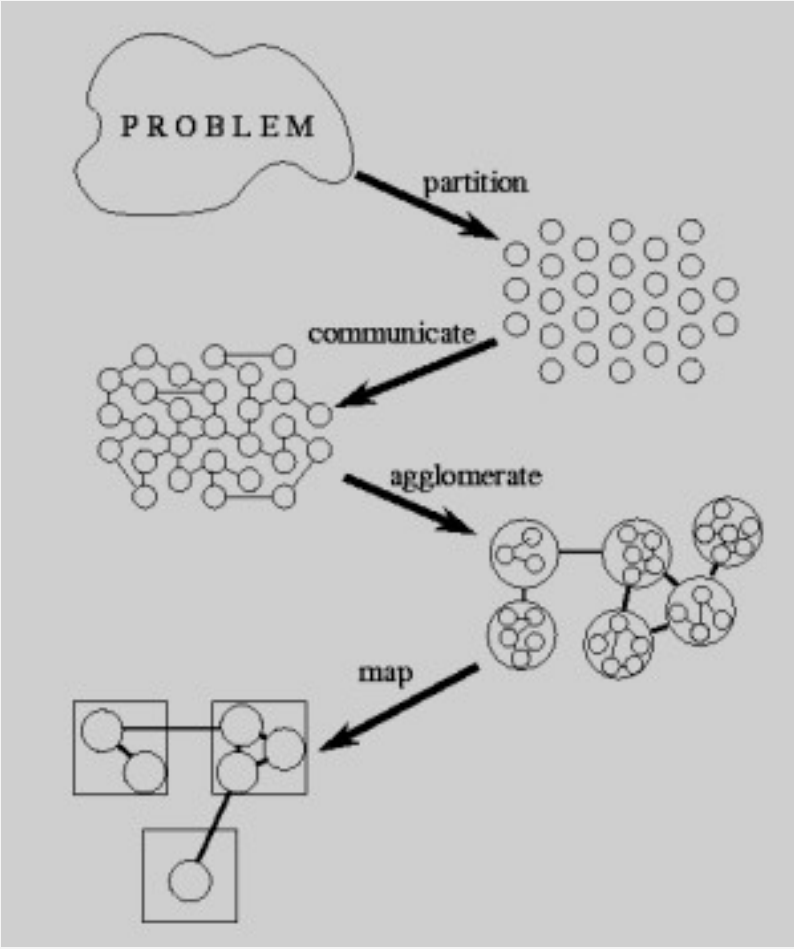
“Invisible” parallelism: exist on every computer

- Bit level parallelism – by **compiler**
  - *e.g., within floating point operations*
- Instruction level parallelism (ILP) – **by compiler/HW**
  - *multiple instructions execute per clock cycle*
- Memory system parallelism – **by compiler/HW**
  - *prefetch, overlap of memory operations with computation*
- Job parallelism – **by OS**
  - *multiple jobs are run in parallel by the OS*

At the parallel application level (parallel tasks and their management), programmers are on their own!

Granularity, mapping

- Partitions = tasks
  - *right size for mapping*
- Communication =>
  - *overhead*
- Data locality =>
  - *agglomerate*
- Mapping =>
  minimize overheads and allow for good load balancing



Beware: agglomerate doesn't always work,
Larger granularity ≠ better data locality

Data locality

- The further the data is from the computation, the slower the computation becomes
- Algorithm should do most work on local data
  - *requires smart data distribution*
- When locality is not possible, use latency hiding techniques
  - *caching and prefetching*
  - *overlapping computation with data transfers*

$$data\ locality\ ratio = \frac{computing\ time}{mem.access \text{ or } communication\ time}$$

Increase data locality by using a data element many times after it is read into cache or communicated from remote (slow) memory.

Challenges

- **Hardware**: processor technology and interconnection network;
- **Software**: Parallel compilers; Grid computing environment (e.g., Condor, Globus, .net, etc.)
- **Methods and Algorithms**: Only computations with large degree of parallelism can efficiently use parallel and distributed computers. (Another important factor is algorithmic efficiency: High performance algorithms are of the same importance as high performance computers!)

Algorithms: Load imbalance

- Workload is imbalanced between processors
  - *less parallelism than processors*
  - *differently sized tasks*
- Can all applications be load-balanced?
  - *adapting to "interesting parts of a domain"*
  - *tree-structured computations*
  - *fundamentally unstructured problems*
- Consequences:
  - *idle resources*
  - *poor hardware utilization*
  - *poor performance*

Lecture 2

Parallel & Distributed Architecture

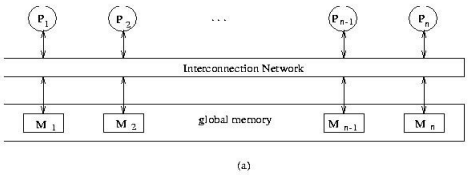
Parallel Computer Architectures

1. Memory organization
   - Memory hierarchy
   - Shared versus distributed memory
2. Processor/node architecture
   - Flynn's taxonomy
3. Interconnection network
   - Dynamic versus static networks

1. Classification based on Memory

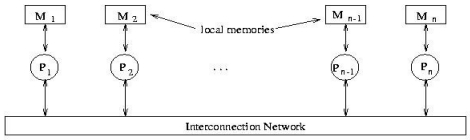
1. Based on memory organization:
   SM=Shared-Memory versus DM=Distributed Memory
2. Based on access time:
   - UMA =Uniform Memory Access (time);
   - NUMA=Non-uniform Memory Access (time);
   - SMP = Symmetric Multi-Processors;

Shared memory



(a)

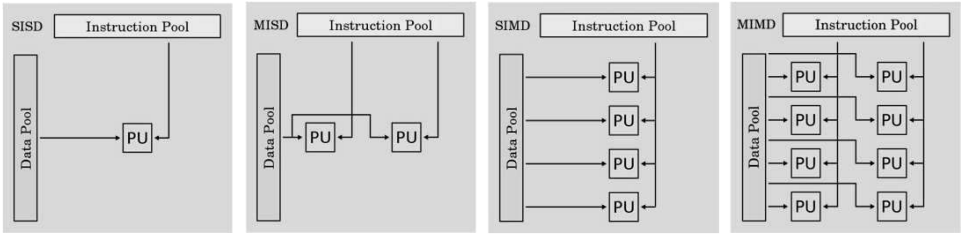
Distributed memory



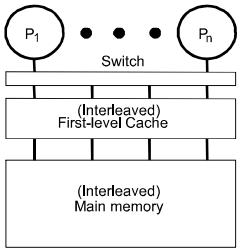
2. Classification based on data and control flows

Flynn's taxonomy:

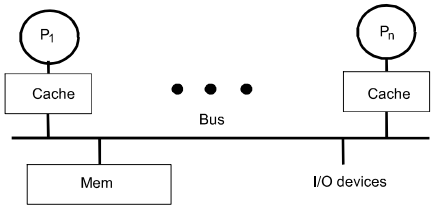
1. Single Instruction Single Data (SISD) – exp. classical Von Neumann machine (sequential computer).
2. Multiple Instruction Single Data (MISD) – exp. none
3. Single Instruction Multiple Data (SIMD) – exp. GPU
4. Multiple Instruction Multiple Data (MIMD) – exp. cluster of computers



Extensions of Memory System (1)

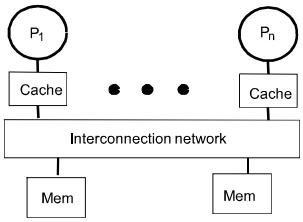


(a) Shared Cache

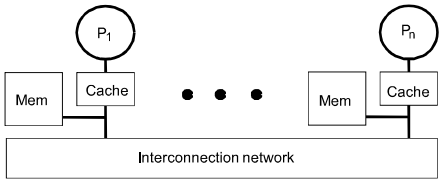


(b) Bus-based shared memory (SMP)

Extensions of Memory System (2)



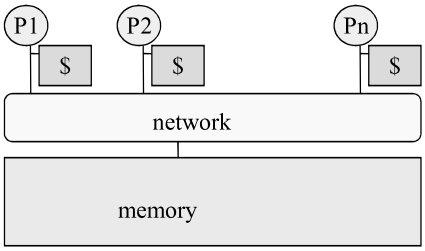
(c) Dance hall (UMA)



(d) Distributed-memory (NUMA)

Machine Model 1

- **A shared memory machine**
- **Processors all connected to a large shared memory**
- **"Local" memory is not (usually) part of the hardware**
  - **Symmetric Mutliprocessors (SMP), e.g. SGI Origin**
- **Speed: much quicker to cache than main memory**



Bus-based Shared Memory Multiprocessors

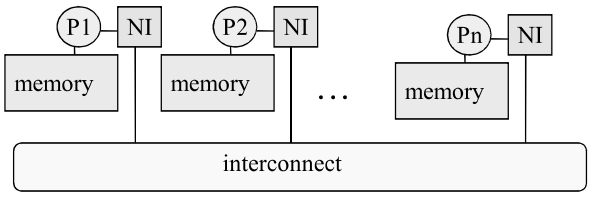
- Symmetric Multiprocessors (SMPs)
  - Symmetric access to all of main memory from any processor
- Dominate the server market
  - Building blocks for larger systems; today multi-core laptops are common
- Attractive as throughput servers and for parallel programs
  - Fine-grain resource sharing
  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Useful for operating system too

Shared Memory Multiprocessors

- Normal uniprocessor mechanisms to access data through reads and writes
- Key is extension of memory hierarchy to support multiple processors (e.g., crossbar network is expensive for large number, therefore often virtual shared memory system is implemented)

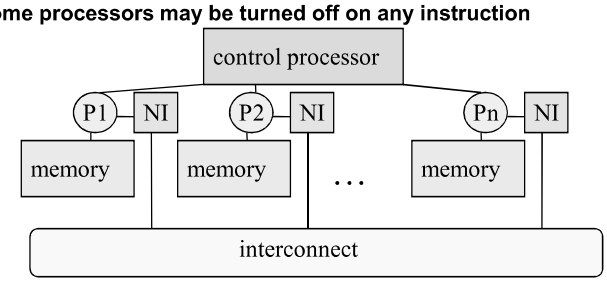
Machine Model 2

- **A distributed memory machine**
  - **Cluster of computers, IBM Blue Gene,Tianhe, etc.**
  - **Processors all connected to own memory (and caches)**
  - **cannot directly access another processor's memory**
- **Each "node" has a network interface (NI)**
  - **all communication and synchronization done through this**



Machine Model 3

- A SIMD (Single Instruction Multiple Data) machine
- A large number of small processors
- A single "control processor" issues each instruction
  - each processor executes the same instruction
  - some processors may be turned off on any instruction



Earlier example machine Connection Machine (CM). Programming model is
- implemented by mapping n-fold parallelism to p processors
- mostly done in the compilers (HPF = High Performance Fortran)

Machine Model 4, Cluster of SMPs

- Since small shared memory machines (SMP's) are the fastest commodity machine, why not build a larger machine by connecting many of them with a network?
- Shared memory within one SMP
- Message passing outside
- ASCI Red (Intel), Blue Gene (IBM), ...
- Programming model?
  - Treat machine as "flat", always use message passing, even within SMP (simple, but ignore important part of memory hierarchy)
  - Expose two layers: shared memory and message passing (higher performance, e.g., mixed MPI&OpenMP, but ugly to program)

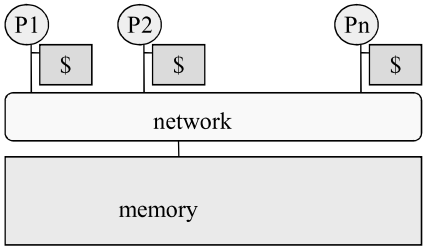
Shared Memory Systems

- Shared cache
- Shared memory
- Cache coherence problem

- Emphasis is not on network, but on memory hierarchy.

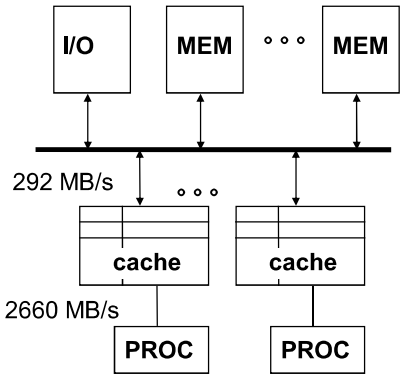
Basic Shared Memory Architecture

- Processors all connected to a large shared memory
- Local caches for each processor
- speed: much quicker to cache than main memory



° Simplest to program, but hard to build with many processors

Limits of using Bus as Network



Assume a 1000 MB/s bus

500 MIPS processor w/o cache

=> 2000 MB/s instr BW per processor

=> 660 MB/s data BW if 33% load-store

(assuming 4 bytes per instruction/word)

Suppose 98% instr. hit rate and 95% data hit rate (assume 16 bytes block=cache line)

=> 160 MB/s instr. BW per processor

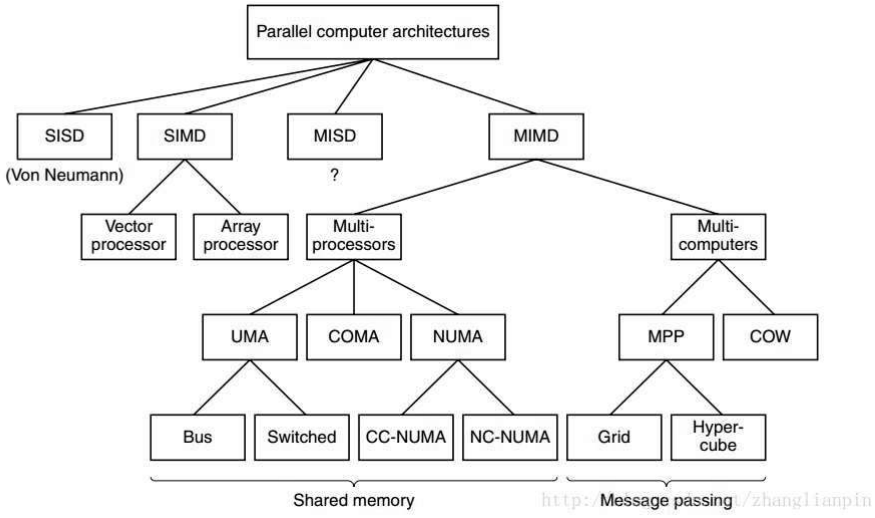
=> 132 MB/s data BW per processor

=> 292 MB/s combined BW

∴ 4 processors will saturate the bus!

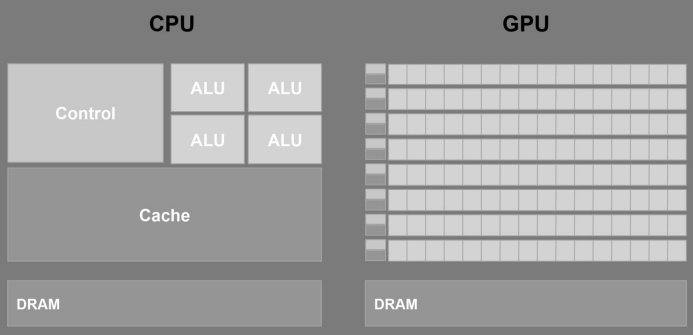
→ **Bus only useful in small systems**

Summary: Parallel Computer Architectures



Accelerators (for floating point processing)

A Graphics Processing Unit is not a CPU!



And there is no GPU without a CPU…

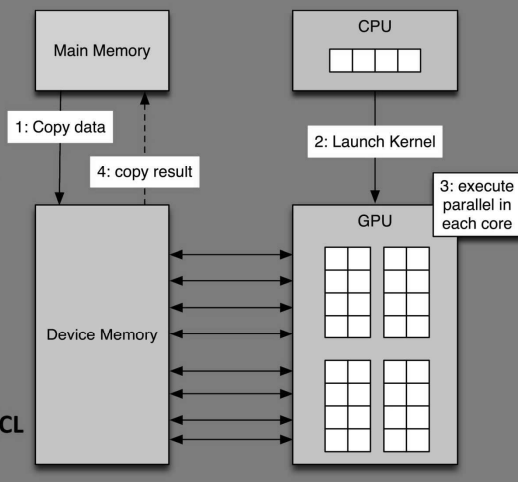
Graphics Processing Unit

✓Originally dedicated to specific operations required by video cards for accelerating graphics, GPU have become flexible **general-purpose** computational engines (GPGPU):

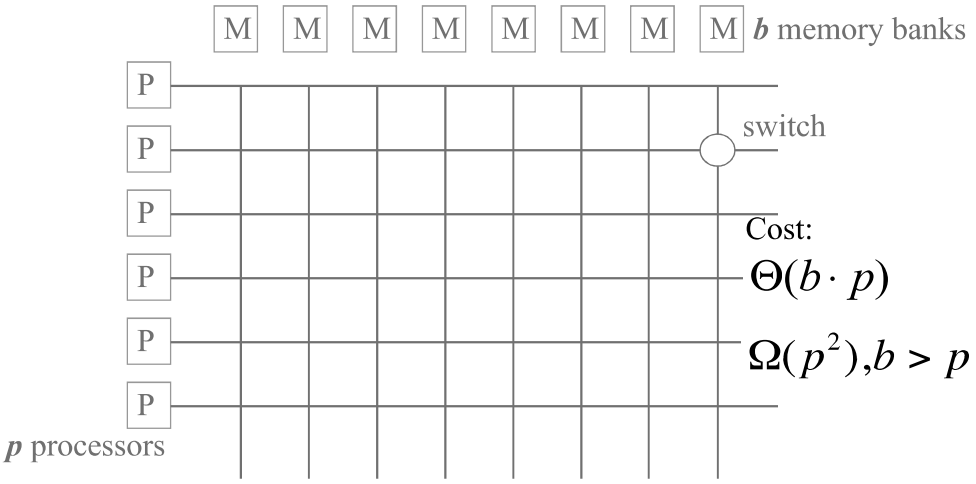
- ✓Optimized for **high memory throughput**
- ✓Very suitable to **data-parallel processing**

✓Three vendors: Chipzilla (*a.k.a.* Intel), ADI (*i.e.,* AMD), NVIDIA.

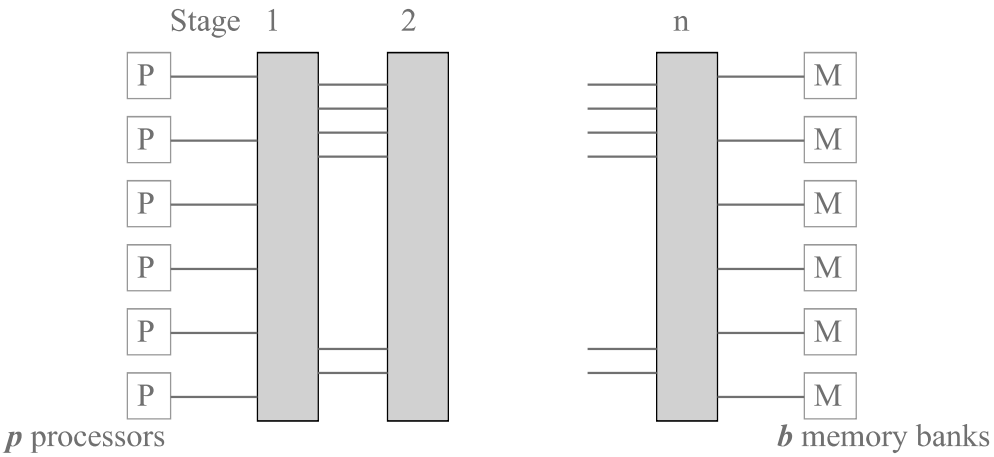
✓Traditionally GPU programming has been tricky but **CUDA** and **OpenCL** made it affordable.



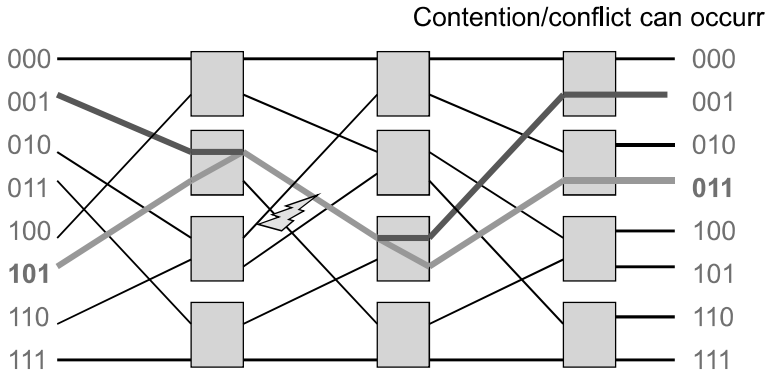
Cross-bar networks (dynamic)



Multi-stage networks (dynamic)



Collision in Omega Network



Exactly one path for every pair: $s = \alpha_1\alpha_2...\alpha_k$ to $t = \beta_1\beta_2...\beta_k$

In total $(n/2)\log(n)$ swtiches $\rightarrow$ $2^{(n/2)\log(n)} = n^{n/2}$ different switchings compared to $n!$ permutations (for n input to n output), only $n^{n/2}$ of the $n!$ possible permutations can be performed without conflict.

Dynamic networks (switching network)

- **Flexible in realizing communication of diff. pairs of processors/memories**
- **Simple bus type network cheap but not scalable (in performance) to large number of processors**
- **Crossbars are very fast but too expensive to scale to large systems**
- **Trade-off between cost and performance**

Network Analogy

- To have a large number of transfers occurring at once, you need a large number of distinct wires
- Networks are like streets
  - link = street
  - switch = intersection
  - distances (hops) = number of blocks traveled
  - routing algorithm = travel plans
- Properties
  - latency: how long to get somewhere in the network
  - bandwidth: how much data can be moved per unit time
    - » limited by the number of wires
    - » and the rate at which each wire can accept data

Components of a Network

Networks are characterized by

- Topology - how things are connected
  - two types of nodes: hosts and switches
- Routing algorithm - paths used
  - e.g., all east-west then all north-south (avoids deadlock)
- Switching strategy
  - circuit switching: full path reserved for entire message
    - » like the telephone
  - packet switching: message broken into separately-routed packets
    - » like the post office
- Flow control - what if there is congestion
  - if two or more messages attempt to use the same channel
  - may stall, move to buffers, reroute, discard, etc.

Properties of a Network

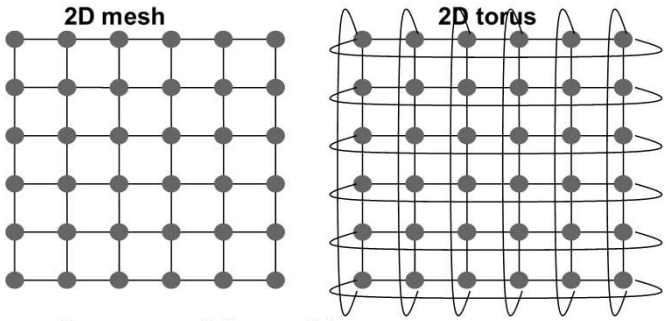
- **Diameter** is the maximal length of shortest paths between any two nodes in the graph. (another metric: average distance)
- The **bandwidth** of a link is: w * 1/t
  - w is the number of wires
  - t is the time per bit
- Effective bandwidth lower due to packet overhead

- **Bisection bandwidth**
  - sum of the minimum number of channels which, if removed, will separate the network into two equal parts

    (A network is partitioned if some nodes cannot reach others.)

Meshes and Tori

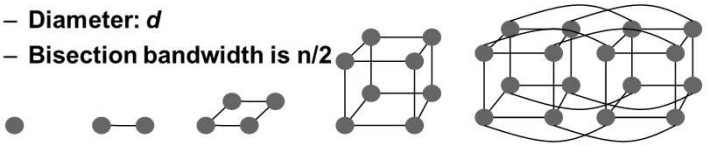
- Diameter: $2\sqrt{n}$ (in 2D)
- Bisection bandwidth: $\sqrt{n}$



- Often used as network in machines
- Generalizes to higher dimensions (Cray T3D used 3D Torus)
- Natural for algorithms with 2D, 3D arrays

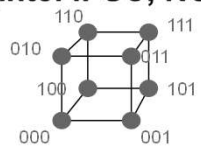
Hypercubes

- Number of nodes n = 2^d for dimension *d*
  - Diameter: *d*
  - Bisection bandwidth is n/2



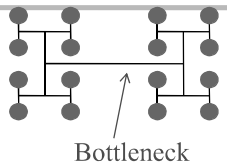
0d 1d 2d 3d 4d

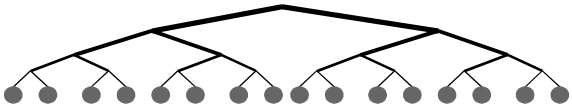
- Popular in early machines (Intel iPSC, NCUBE)
  - Lots of clever algorithms
- Greycode addressing
  - each node connected to d others with 1 bit different

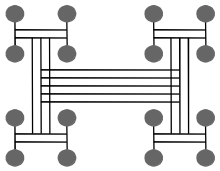


Trees

- Diameter: log(n)
- Bisection bandwidth: 1
- Easy layout as planar graph
- Many tree algorithms (summation)
- Fat trees avoid bisection bandwidth problem
  - more (or wider) links near top
  - example, Thinking Machines CM-5



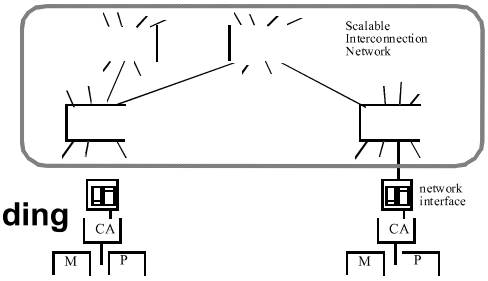




Scalable, High Perf. Interconnection Network

- At the core of parallel computer architecture
- Requirements and trade-offs at many levels
  - Elegant mathematical structure
  - Deep relationships to algorithm structure
  - Managing many traffic flows
  - Electrical / Optical link properties
- Little consensus
  - interactions across levels
  - Performance metrics?
  - Cost metrics?
  - Workload?

=> need a holistic understanding



Example speccification: Summit supercomputer

Processors:
4,356 nodes, each with two 22-core Power9 CPUs, and six NVIDIA Tesla V100 GPUs. Total 2,414,592 cores

A Fat-tree network topology
InfiniBand uses a switched fabric topology, as opposed to early shared medium Ethernet.
Implemented using Mellanox 100-Gb/s EDR InfiniBand ConnectX-5 adapters and Switch-IB2 switches
Messages
InfiniBand transmits data in packets of up to 4 KB that are taken together to form a message.

Distributed Memory

→ Each processing element (P) has its separate main memory block (M)

→ Data exchange is achieved through message passing over the network

→ Message passing could be either explicit (MPI) or implicit (PGAS)

→ Programs typically implemented as a set of OS entities with own (virtual) address spaces – *processes*

→ No shared variables

→No data races

→Explicit synchronisation mostly unneeded

→Results as side effect of the send-receive semantics

Processes

- **A process is a running in-memory instance of an executable file**
  - → Executable code, e.g., binary machine instructions
  - → One or more threads of execution sharing memory address space
  - → Memory: data, heap, stack, processor state (CPU registers and flags)
  - → Operating system context (e.g. signals, I/O handles, etc.)
  - → PID
- **Isolation and protection**
  - → A process cannot interoperate with other processes or access their context (even on the same node) without the help of the operating system
  - → No direct inter-process data exchange (isolated/virtual address spaces)
  - → No direct inter-process synchronisation

General Structure of an MPI Program

- **Start-up, initialisation, finalisation, and shutdown – C**

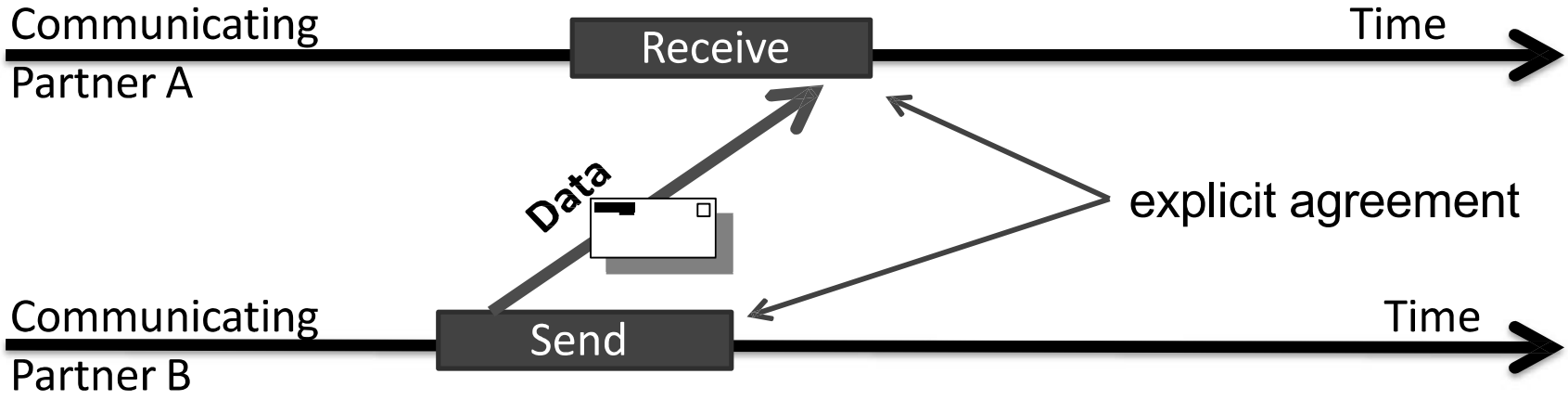
C

```
1  #include <mpi.h>

   int main(int argc, char **argv)
2  {
     … some code …
3    MPI_Init(&argc, &argv);



4      … computation & communication …



5    MPI_Finalize();
     … wrap-up …
6    return 0;
   }
```

1. Inclusion of the MPI header file
2. Pre-initialisation mode: uncoordinated
   - **No MPI function calls allowed with few exceptions**
   - **All program instances run exactly the same code**
3. Initialisation of the MPI environment
   Implicit synchronisation
4. Parallel MPI code Typically computation and communication
5. Finalisation of the MPI environment
   Internal buffers are flushed
6. Post-finalisation mode: uncoordinated
   - **No MPI function calls allowed with few exceptions**

Message Passing

- **The goal is to enable communication between processes that share no memory space**



- **Explicit message passing requires:**
  - → Send and receive primitives (operations)
  - → Known addresses of both the sender and the receiver
  - → Specification of what has to be sent/received

MPI Datatypes

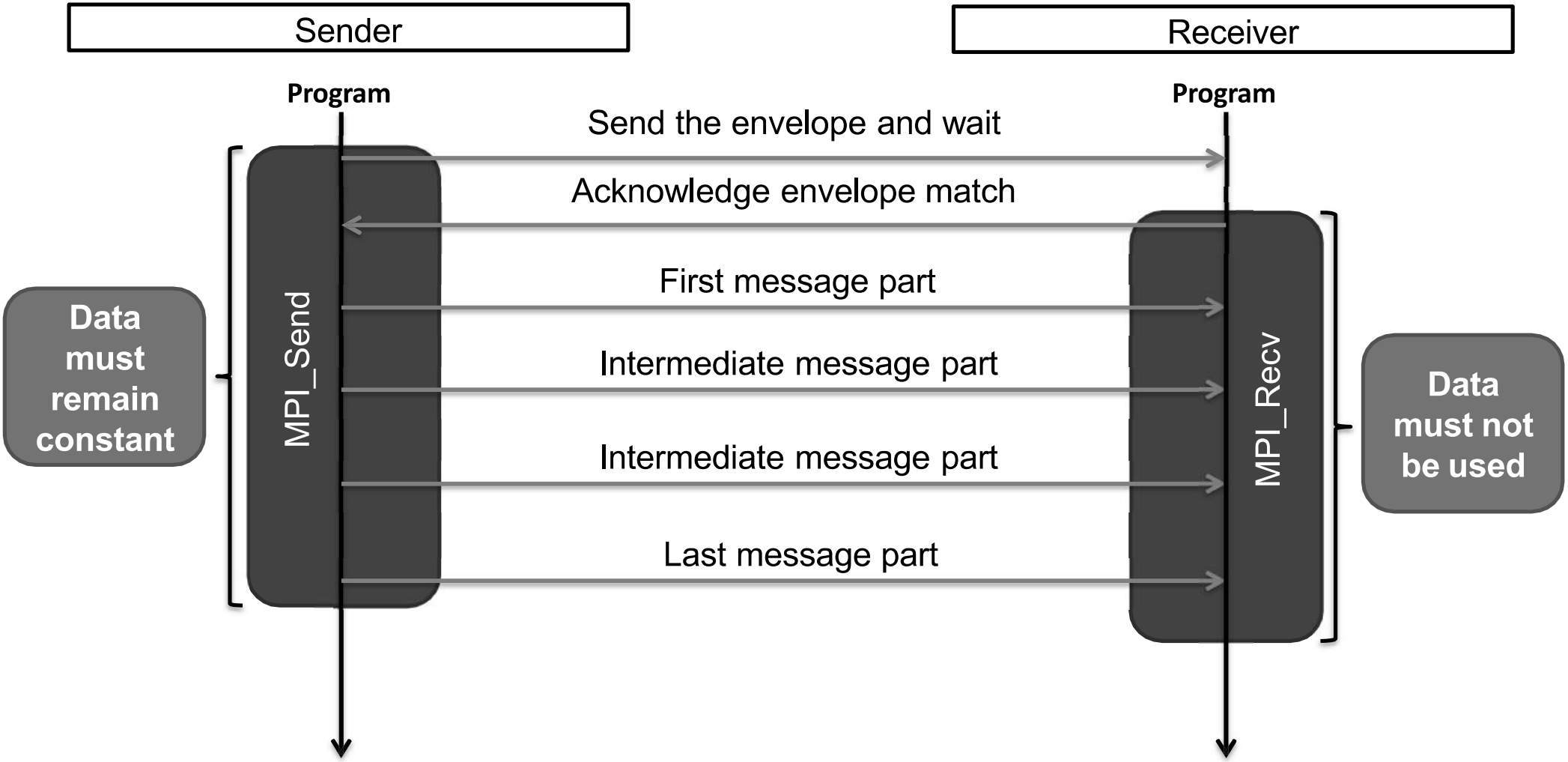
- **MPI is a library – it cannot infer the type of elements in the supplied buffer at run time and that's why it has to be told what it is**

- **MPI datatypes tell MPI how to:**
  - → read binary values from the send buffer
  - → write binary values into the receive buffer
  - → correctly apply value alignments
  - → convert between machine representations in heterogeneous environments

- **MPI datatype must match the language type(s) in the data buffer**
- **MPI datatypes are handles and cannot be used to declare variables**

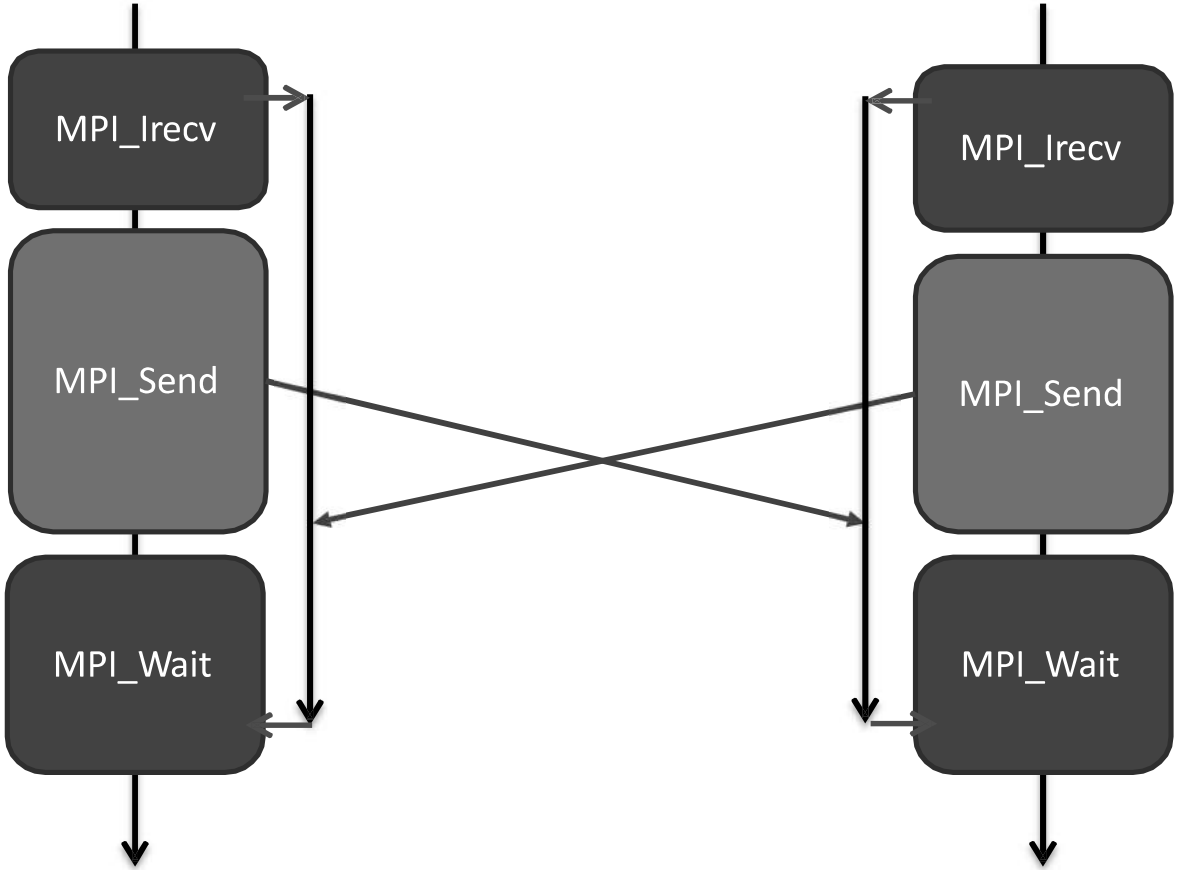
Blocking Calls

- **Blocking send (w/o buffering) and receive calls:**



Deadlock Prevention

- **Non-blocking operations can be used to prevent deadlocks in symmetric code:**



- **That is how MPI_Sendrecv is usually implemented**

Send Modes

- **Standard mode**
  - → The call blocks until the message has either been transferred or copied to an internal buffer for later delivery
- **Synchronous mode**
  - → The call blocks until a matching receive has been posted and the message reception has started
- **Buffered mode**
  - → The call blocks until the message has been copied to a user-supplied buffer. Actual transmission may happen at a later point
- **Ready mode (don't use!)**
  - → The operation succeeds only if a matching receive has already been posted. Behaves as standard send in every other aspect

Send Modes

- **Call names:**
  - → **MPI_Send** blocking standard send
  - → **MPI_Isend** non-blocking standard send
  - → **MPI_Ssend** blocking synchronous send
  - → **MPI_Issend** non-blocking synchronous send
  - → **MPI_Bsend** blocking buffered send
  - → **MPI_Ibsend** non-blocking buffered send
  - → **MPI_Rsend** blocking ready-mode send
  - → **MPI_Irsend** non-blocking ready-mode send
- **Buffered operations require an explicitly provided user buffer**
  - → **MPI_Buffer_attach (void *buf, int size)**
  - → **MPI_Buffer_detach (void *buf, int *size)**
  - → Buffer size must account for the envelope size (**MPI_BSEND_OVERHEAD**)

Common Pitfalls – C/C++

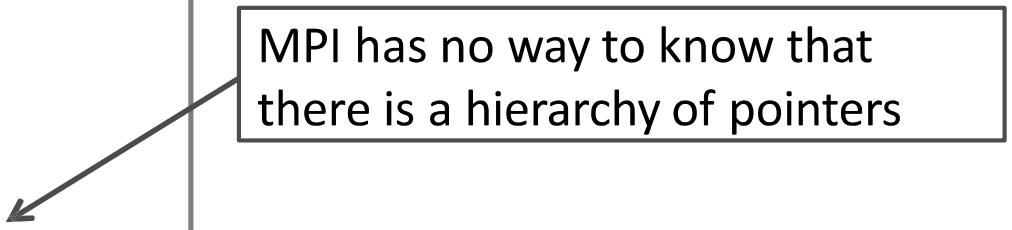
- **Do not pass pointers to pointers in MPI calls**

```
void func (int scalar)
{
  MPI_Send(&scalar, MPI_INT, 1, …

void func (int& scalar)
{
  MPI_Send(&scalar, MPI_INT, 1, …

void func (int *scalar)
{
  MPI_Send(scalar, MPI_INT, 1, …

void func (int *array)
{
  MPI_Send(array, MPI_INT, 5, …
  … or …
  MPI_Send(&array[0], MPI_INT, 5, …
```

Common Pitfalls – C/C++

- **Use flat multidimensional arrays; arrays of pointers do not work**

```
// Static arrays are OK
int mat2d[10][10];
MPI_Send(&mat2d, MPI_INT, 10*10, …

// Flat dynamic arrays are OK
int *flat2d = new int[10*10];
Fill array flat2d …
MPI_Send(flat2d, MPI_INT, 10*10, …

// DOES NOT WORK
int **p2d[10] = new int*[10];
for (int i = 0; i < 10; i++)
   p2d[i] = new int[10];
MPI_Send(p2d, MPI_INT, 10*10, …
… or …
MPI_Send(&p2d[0][0], MPI_INT, 10*10, …
```



Message Passing: Summary

- No notion of global data
- Data communication is done by explicit message passing
  - expensive performance-wise
- Trade-off between:
  - one-copy data
    - *more communication is needed, less consistency issues*
  - local data replication
    - *less communication, consistency is problematic*
- Techniques to improve performance:
  - replicate read-only data
  - computation and communication overlapping
  - message aggregation

MPI Quick Reference in C

```
#include <mpi.h>
```

Environmental Management:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize(void)
int MPI_Initialized(int *flag)
int MPI_Finalized(int *flag)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Abort(MPI_Comm comm, int errorcode)
double MPI_Wtime(void)
double MPI_Wtick(void)
```

Blocking Point-to-Point-Communication:

```
int MPI_Send (const void* buf, int count,
    MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm)
```

Related: `MPI_Bsend, MPI_Ssend, MPI_Rsend`

```
int MPI_Recv (void* buf, int count,
    MPI_Datatype datatype, int source, int
    tag, MPI_Comm comm, MPI_Status *status)
int MPI_Probe (int source, int tag, MPI_Comm
    comm, MPI_Status *status)
int MPI_Get_count (const MPI_Status *status,
    MPI_Datatype datatype, int *count)
```

Related: `MPI_Get_elements`

```
int MPI_Sendrecv (const void *sendbuf, int
    sendcount, MPI_Datatype sendtype, int
    dest, int sendtag, void *recvbuf, int
    recvcount, MPI_Datatype recvtype, int
    source, int recvtag, MPI_Comm comm,
    MPI_Status *status)
int MPI_Sendrecv_replace (void *buf, int
    count, MPI_Datatype datatype, int dest,
    int sendtag, int source, int recvtag,
    MPI_Comm comm, MPI_Status *status)
int MPI_Buffer_attach (void *buffer, int size)
int MPI_Buffer_detach (void *buffer_addr, int
    *size)
```

Non-Blocking Point-to-Point-Communication:

```
int MPI_Isend (const void* buf, int count,
    MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm, MPI_Request *request)
```

Related: `MPI_Ibsend, MPI_Issend, MPI_Irsend`

```
int MPI_Irecv (void* buf, int count,
    MPI_Datatype datatype, int source, int
    tag, MPI_Comm comm, MPI_Request *request)
int MPI_Iprobe (int source, int tag, MPI_Comm
    comm, int *flag, MPI_Status *status)
int MPI_Wait (MPI_Request *request,
    MPI_Status *status)
int MPI_Test (MPI_Request *request, int
    *flag, MPI_Status *status)
int MPI_Waitall (int count, MPI_Request
    request_array[], MPI_Status
    status_array[])
```

Related: `MPI_Testall`

```
int MPI_Waitany (int count, MPI_Request
    request_array[], int *index, MPI_Status
    *status)
```

Related: `MPI_Testany`

```
int MPI_Waitsome (int incount, MPI_Request
    request_array[], int *outcount, int
    index_array[], MPI_Status status_array[])
```

Related: `MPI_Testsome,`

```
int MPI_Request_free (MPI_Request *request)
```

Related: `MPI_Cancel`

```
int MPI_Test_cancelled (const MPI_Status
    *status, int *flag)
```

Collective Communication:

```
int MPI_Barrier (MPI_Comm comm)
int MPI_Bcast (void *buffer, int count,
    MPI_Datatype datatype, int root, MPI_Comm
    comm)
int MPI_Gather (const void *sendbuf, int
    sendcount, MPI_Datatype sendtype, void
    *recvbuf, int recvcount, MPI_Datatype
    recvtype, int root, MPI_Comm comm)
int MPI_Gatherv (const void *sendbuf, int
    sendcount, MPI_Datatype sendtype, void
    *recvbuf, const int recvcount_array[],
    const int displ_array[], MPI_Datatype
    recvtype, int root, MPI_Comm comm)
int MPI_Scatter (const void *sendbuf, int
    sendcount, MPI_Datatype sendtype, void
    *recvbuf, int recvcount, MPI_Datatype
    recvtype, int root, MPI_Comm comm)
int MPI_Scatterv (const void *sendbuf, const
    int sendcount_array[], const int
    displ_array[], MPI_Datatype sendtype, void
    *recvbuf, int recvcount, MPI_Datatype
    recvtype, int root, MPI_Comm comm)
int MPI_Allgather (const void *sendbuf, int
    sendcount, MPI_Datatype sendtype, void
    *recvbuf, int recvcount, MPI_Datatype
    recvtype, MPI_Comm comm)
```

Related: `MPI_Alltoall`

```
int MPI_Allgatherv (const void *sendbuf, int
    sendcount, MPI_Datatype sendtype, void
    *recvbuf, const int recvcount_array[],
    const int displ_array[], MPI_Datatype
    recvtype, MPI_Comm comm)
```

Related: `MPI_Alltoallv`

```
int MPI_Reduce (const void *sendbuf, void
    *recvbuf, int count, MPI_Datatype datatype,
    MPI_Op op, int root, MPI_Comm comm)
int MPI_Allreduce (const void *sendbuf, void
    *recvbuf, int count, MPI_Datatype
    datatype, MPI_Op op, MPI_Comm comm)
```

Related: `MPI_Scan, MPI_Exscan`

```
int MPI_Reduce_scatter (const void *sendbuf,
    void *recvbuf, const int
    recvcount_array[], MPI_Datatype datatype,
    MPI_Op op, MPI_Comm comm)
int MPI_Op_create (MPI_User_function *func,
    int commute, MPI_Op *op)
int MPI_Op_free (MPI_Op *op)
```

Derived Datatypes:

```
int MPI_Type_commit (MPI_Datatype *datatype)
int MPI_Type_free (MPI_Datatype *datatype)
int MPI_Type_contiguous (int count,
    MPI_Datatype oldtype, MPI_Datatype
    *newtype)
```

int **MPI_Type_vector** (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

int **MPI_Type_indexed** (int count, const int blocklength_array[], const int displ_array[], MPI_Datatype oldtype, MPI_Datatype *newtype)

int **MPI_Type_create_struct** (int count, const int blocklength_array[], const MPI_Aint displ_array[], const MPI_Datatype oldtype_array[], MPI_Datatype *newtype)

int **MPI_Type_create_subarray** (int ndims, const int size_array[], const int subsize_array[], const int start_array[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

int **MPI_Get_address** (const void *location, MPI_Aint *address)

int **MPI_Type_size** (MPI_Datatype *datatype, int *size)

int **MPI_Type_get_extent** (MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)

int **MPI_Pack** (const void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outcount, int *position, MPI_Comm comm)

int **MPI_Unpack** (const void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

int **MPI_Pack_size** (int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)

Related: MPI_Type_create_hvector, MPI_Type_create_hindexed, MPI_Type_create_indexed_block, MPI_Type_create_darray, MPI_Type_create_resized, MPI_Type_get_true_extent, MPI_Type_dup, MPI_Pack_external, MPI_Unpack_external, MPI_Pack_external_size

Groups and Communicators:

int **MPI_Group_size** (MPI_Group group, int *size)

int **MPI_Group_rank** (MPI_Group group, int *rank)

int **MPI_Comm_group** (MPI_Comm comm, MPI_Group *group)

int **MPI_Group_translate_ranks** (MPI_Group group1, int n, const int ranks1[], MPI_Group group2, const int ranks2[])

int **MPI_Group_compare** (MPI_Group group1, MPI_Group group2, int *result)

MPI_IDENT, MPI_COMGRUENT, MPI_SIMILAR, MPI_UNEQUAL

int **MPI_Group_union** (MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

Related: MPI_Group_intersection, MPI_Group_difference

int **MPI_Group_incl** (MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)

Related: MPI_Group_excl

int **MPI_Comm_create** (MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

int **MPI_Comm_compare** (MPI_Comm comm1, MPI_Comm comm2, int *result)

MPI_IDENT, MPI_COMGRUENT, MPI_SIMILAR, MPI_UNEQUAL

int **MPI_Comm_dup** (MPI_Comm comm, MPI_Comm *newcomm)

int **MPI_Comm_split** (MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

int **MPI_Comm_free** (MPI_Comm *comm)

Topologies:

int **MPI_Dims_create** (int nnodes, int ndims, int dims[])

int **MPI_Cart_create** (MPI_Comm comm_old, int ndims, const int dims[], const int periods[], int reorder, MPI_Comm *comm_cart)

int **MPI_Cart_shift** (MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)

int **MPI_Cartdim_get** (MPI_Comm comm, int *ndim)

int **MPI_Cart_get** (MPI_Comm comm, int maxdims, int dims[], int periods[], int coords[])

int **MPI_Cart_rank** (MPI_Comm comm, const int coords[], int *rank)

int **MPI_Cart_coords** (MPI_Comm comm, int rank, int maxdims, int coords[])

int **MPI_Cart_sub** (MPI_Comm comm_old, const int remain_dims[], MPI_Comm *comm_new)

int **MPI_Cart_map** (MPI_Comm comm_old, int ndims, const int dims[], const int periods[], int *new_rank)

int **MPI_Graph_create** (MPI_Comm comm_old, int nnodes, const int index[], const int edges[], int reorder, MPI_Comm *comm_graph)

int **MPI_Graph_neighbors_count** (MPI_Comm comm, int rank, int *nneighbors)

int **MPI_Graph_neighbors** (MPI_Comm comm, int rank, int maxneighbors, int neighbors[])

int **MPI_Graphdims_get** (MPI_Comm comm, int *nnodes, int *nedges)

int **MPI_Graph_get** (MPI_Comm comm, int maxindex, int maxedges, int index[], int edges[])

int **MPI_Graph_map** (MPI_Comm comm_old, int nnodes, const int index[], const int edges[], int *new_rank)

int **MPI_Topo_test** (MPI_Comm comm, int *status)

Wildcards:

MPI_ANY_TAG, MPI_ANY_SOURCE

Basic Datatypes:

MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED

Predefined Groups and Communicators:

MPI_GROUP_EMPTY, MPI_GROUP_NULL, MPI_COMM_WORLD, MPI_COMM_SELF, MPI_COMM_NULL

Reduction Operations:

MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND, MPI_LOR, MPI_LXOR

Status Object:

status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR

Complexity of Algorithms; 2D Poisson with *N* unknowns

Algorithm	Serial	PRAM	Memory	#Procs
° Dense LU	N^3	N	N^2	N^2
°				
° Jacobi	N^2	N	N	N
° RB SOR	$N^{3/2}$	$N^{1/2}$	N	N
° Conj.Grad.	$N^{3/2}$	$N^{1/2}$ *log N	N	N
° FFT	N*log N	log N	N	N
° Multigrid	N	$\log^2 N$	N	N
° Lower bound	N	log N	N	

PRAM is an idealized parallel model with zero cost communication

Jacobi's Method

- To derive Jacobi's method, write Poisson (2D) as:

$$u(i,j) = (u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) + b(i,j))/4$$

- Let $u^m(i,j)$ be approximation for u(i,j) after m steps

$$u^{m+1}(i,j) = (u^m(i-1,j) + u^m(i+1,j) + u^m(i,j-1) + u^m(i,j+1) + b(i,j)) / 4$$

- I.e., $u^{m+1}(i,j)$ is a weighted average of its neighbors
- Motivation: $u^{m+1}(i,j)$ chosen to exactly satisfy the discretized equation at (i,j)
- Convergence is proportional to problem size, $N=n^2$
- Therefore: serial complexity, number of iterations times work per iteration, is $O(N^2)$.

Successive Overrelaxation (SOR)

- Red-black Gauss-Seidel converges twice as fast as Jacobi, but there are twice as many parallel steps
- To motivate next improvement, rewrite basic step in algorithm :

$$u^{m+1}(i,j) = u^{m}(i,j) + correction^{(m)}(i,j)$$

- then one should move even further in that direction If "correction" is a good direction to move, i.e., w>1

$$u^{m+1}(i,j) = u^{m}(i,j) + w * correction^{(m)}(i,j)$$

- It's called successive overrelaxation (SOR)
  - Successive: always use latest information like in ordinary Gauss-Seidel
  - Overrelaxtion: w > 1
  - But w <2 is necessary for convergence

Red-Black SOR

- **Parallelizes like Jacobi**
  - **Still sparse-matrix-vector multiply…**
- **Can be proved: w = 2/(1+sin(π/(n+1))) for best convergence**
  - **Number of steps to converge = parallel complexity = O(n), instead of $O(n^2)$ for Jacobi**
  - **Serial complexity $O(n^3) = O(N^{3/2})$, instead of $O(n^4) = O(N^2)$ for Jacobi.**
- **In general w_{Opt} behaves as 2 - O(1/n).**
- **For general matrices T, w_{Opt} should be determined empirically.**

Parallel time complexity: minimum parallel execution time of an algorithm on a PRAM computer (see extra slides at the end) with infinite many processors and zero communication cost.
(So it only considers parallelism (parallel operations) but ignores communication overhead).

Conjugate Gradient (CG) for solving A*x = b

- **This method can be used when the matrix A is**
  - **symmetric, i.e., $A = A^T$**
  - **positive definite, defined equivalently as:**
    - **all eigenvalues are positive**
    - **$x^T * A * x > 0$ for all nonzero vectors s**
    - **a Cholesky factorization, $A = L*L^T$ exists**
- **Algorithm maintains 3 vectors**
  - **x = the approximate solution, improved after each iteration**
  - **r = the residual, r = b- A*x**
  - **p = search direction, also called the conjugate gradient**

There are a number of different computations in a CG-iteration, where should we begin with the parallelization?

➔ Rule of thump: start with the most compute-intensive/communication-intensive step.

Conjugate Gradient; computation/operations

° **Algorithm maintains 3 vectors**

- **$\underline{x}$ = the approximate solution, improved after each iteration**
- **$\underline{r}$ = the residual, $\underline{r} = \underline{b} - A*\underline{x}$**
- **$\underline{p}$ = search direction, also called the conjugate gradient**

° **Start with**

- **$\underline{x}=\underline{0}$, $r=\underline{b}$, $\underline{p}=\underline{b}$**

° **Iterate until $\underline{r}\cdot\underline{r}$ is small enough**

- **$\underline{v} = A \cdot \underline{p}$** Matrix-vector multiplication
- **$a = (\underline{r} \cdot \underline{r}) / (\underline{p} \cdot \underline{v})$** Inner product (1x)
- **$\underline{x} = \underline{x} + a * \underline{p}$** vector +scalar*vector
- **$\underline{r}_{old} = \underline{r}$** copy
- **$\underline{r} = \underline{r} - a * \underline{v}$** vector - scalar*vector
- **$\underline{p} = \underline{r} + (\underline{r} \cdot \underline{r}) / \underline{r}_{old} \cdot \underline{r}_{old}) \underline{p}$** vector + scalar*vector

Complexity of Conjugate Gradient (CG)

- **One iteration costs**
  - **Sparse-matrix-vector multiply by A (major cost)**
  - **3 dot products, 3 *saxpys* (scalar*vector + vector)**

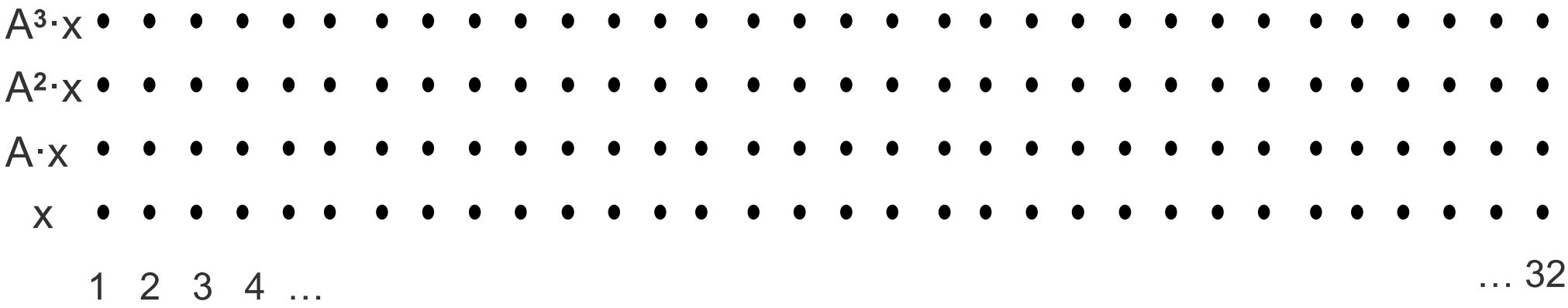
- **Converges in O(n) = O($N^{1/2}$) steps, like SOR**
  - **Serial complexity = O($N^{3/2}$)**
  - **Parallel complexity = O($N^{1/2}$ log N),**
  - **The log N factor is from dot-products. Global sum needs to be done. This can be obtained by adding the N (single) products in log N phases.**

- **Implementation on a real parallel computer, computing inner products can be the dominant communication overhead. Why?**

Local (neighbour) communication versus global communication

Communication Avoiding Jacobi:

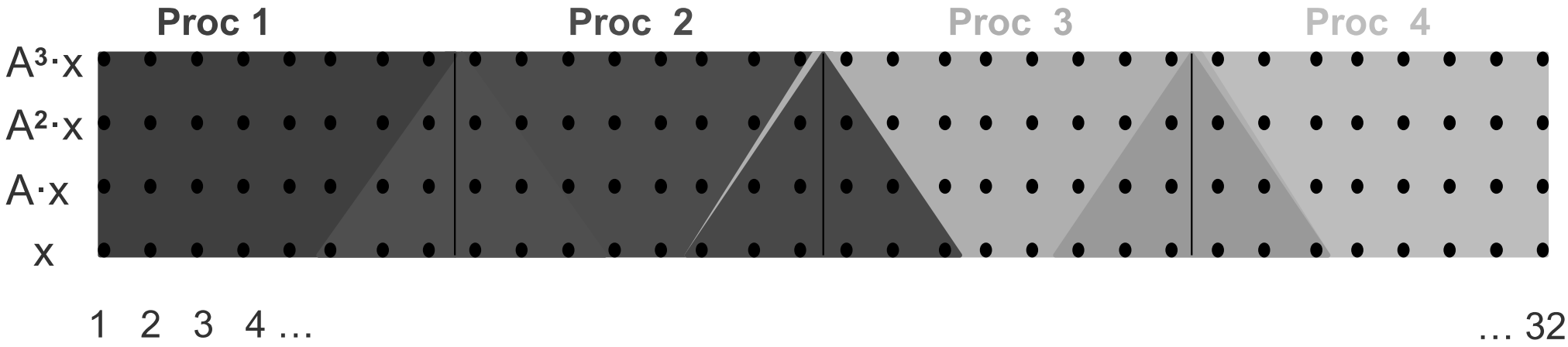
- Iterative methods require exchange of data of neighbor grid points after every iteration. How to reduce the communication overhead? The following is one way of optimization.
- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$



- Example: A tridiagonal, n=32, k=3 (1-D problem). Simplified here: compared to slide 4, A=(T/2-I) and $x^{(m+1)} = Ax^{(m)}+b/2$.
- Like computing the powers of the matrix, but simpler:
  - Don't need to store A explicitly (it's Jacobi)
  - Only need to save vectors A^ix for i=1,…,k

Communication Avoiding Jacobi:

- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]
- Parallel Algorithm: perform k iterations at once before the next communication step



- Example: A tri-diagonal, n=32, k=3
- Trade-off:
  - Entries in overlapping regions (triangles) computed redundantly
  - Send O(1) messages instead of O(k)

Summary of Jacobi, SOR and CG

- Jacobi, SOR, and CG all perform sparse-matrix-vector multiply
- For Poisson, this means nearest neighbor communication on an n-by-n grid ($N=n^2$)
- Parallelization with Red-Black ordering: decoupling dependence.
- Optimization of communication with performing multiple iterations at once (comm. avoidance)
- Limitations of Jacobi, SOR and CG methods:
  - It takes $n = N^{1/2}$ steps for information to travel across an n-by-n grid.
  - Since the solution on one side of grid depends on data on other side of grid faster methods require faster ways to move information
    - Multigrid (next lecture)
    - FFT

Parallel computing on Graphics Cards 2

Advantages

- Hardware is (was ?) cheap compared with workstations clusters
- Lower power consumption per Flop than CPU
- Simple GPU already inside many systems without extra investment
- Capable of thousands of parallel threads on a single GPU card
- Very fast for algorithms that can be efficiently parallelised
- Often better scalability than MPI for large problems due to faster communication
- New libraries hiding complexity: **BLAS, FFTW, SPARSE, SOLVER**
- Easy integration with 3D graphics if this runs on the same card (or by using Prime)

Parallel computing on Graphics Cards 3

Disadvantages

- Limited amount of memory available (between 2-16 GByte)
- Memory transfers between host and graphics card add extra overhead
- Often no ECC : error correcting memory (except highend GPUs)
- Fast double precision GPUs still quite expensive
- Slow for algorithms without enough data parallellism
- Debugging code on GPU can be complicated
- Achieving theoretically possible speedup almost impossible
- Using Multi GPUs in a cluster is complex (often with MPI or OpenMP)

What is Cuda ?

means "Compute Unified Device Architecture" and is a software toolkit by Nvidia to ease the use of (Nvidia) graphics cards for scientific programming. It needs on a special video driver as well.

Features :

- Special C compiler to build code both for CPU and GPU (nvcc)
- C Language extensions (codewords) :
  - distinguish CPU and GPU functions
  - access different types of memory on the GPU
  - specify how code should be parallelized on the GPU
- Library routines for memory transfer between host and GPU
- Extra BLAS, Sparse and FFT libraries for easy porting existing code
- Dedicated to Nvidia hardware
- Mainly standard C on the GPU (limited support for C++ and Fortran)

What is OpenCL ?

means "Open Computing Language" and is a C99 like language for parallel kernels in combination with an API to get these kernels loaded and started on dedicated parallel hardware (Apple, Khronos, Nvidia).

Features :

- Only contains a dedicated compiler for building GPU code
- So, no support for mixed (GPU + CPU) source code as in Cuda
- Data transfer between host and GPU done using library functions
- Runs on several types of GPU (AMD, Nvidia) and is easy portable
- However, performance is not necessarily portable across platforms
- OpenCL is an Open Standard, Cuda causes a vendor-lock to Nvidia
- Less mature than Cuda, often no support for latest GPU features
- Starting kernels more complicated than Cuda

When to use GPUs ? FFT comparison

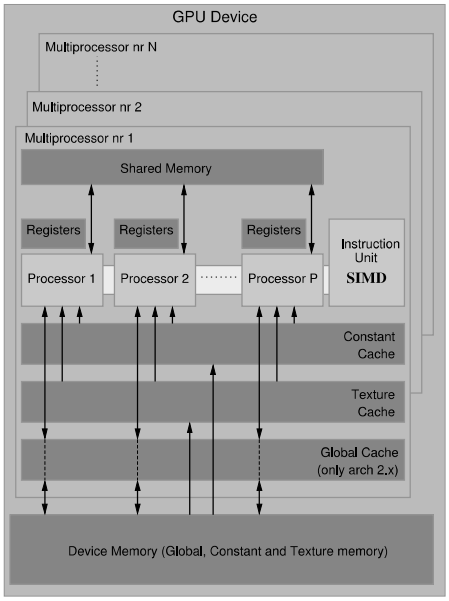
Let's also compare Fast Fourier Transforms on some platforms. For CPUs we used **FFTW** and for GPUs **CuFFT**. Time in **milli**seconds.

Type (nc=no communication)	512	4096	32768	262144	2097152	16777216
CPU Core Duo 1x	0.21	0.35	1.6	16	178	1710
CPU Xeon Oct 1x	0.21	0.34	1.6	15	175	1647
CPU Xeon Oct 50x	0.34	2.60	39.4	477	7040	69790
K600 1x	0.1	0.1	0.6	2.5	15	117
K600 50x (nc)	1.2	1.3	5.2	39.1	243	2002
TitanX 1x	0.2	0.2	0.6	3.9	27	208
TitanX 50x (nc)	1.9	2.3	3.1	5.1	25	176
CPU/GPU 1x TitanX	1x	1.5x	3x	4x	7x	8x
CPU/GPU 50x TitanX (nc)	0.2x	1.1x	13x	94x	280x	397x

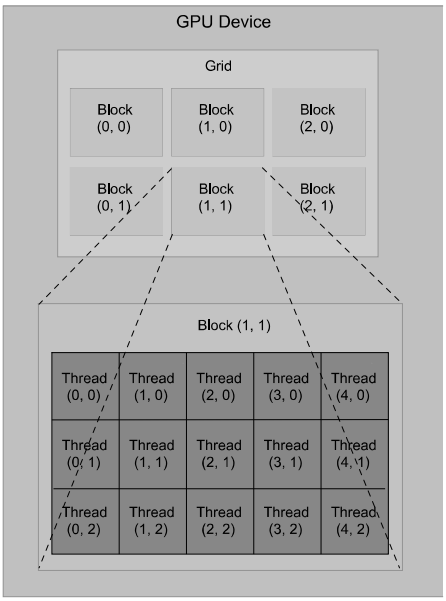
Conclusions :

- Old CPU Core Duo (2010) as fast as an Oct core Xeon (2016) ...
- For small problems cheap GPUs are faster than large GPUs (clock).
- GPU transfer time increases with size, computation time only when full occupation is reached.

Hardware layout of the GPU



Processing layout of a GPU kernel



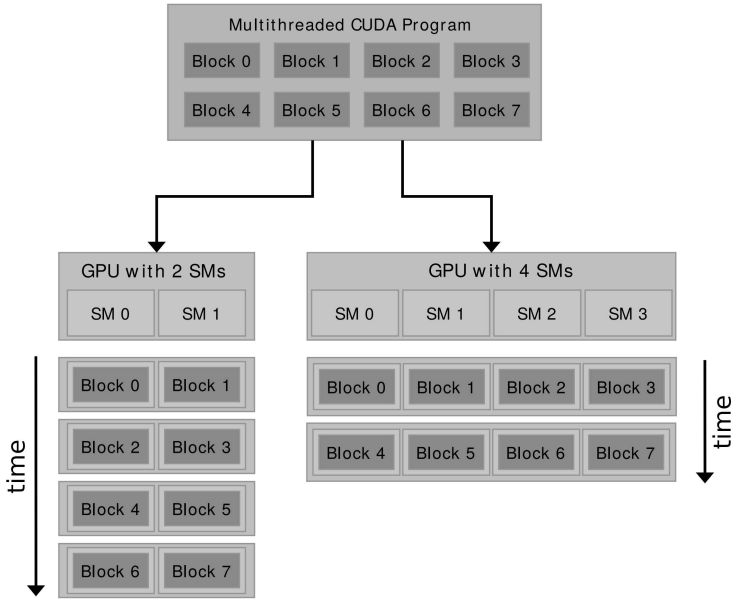
Relation software $\Longleftrightarrow$ hardware

Thread $\Longleftrightarrow$ **Core**

Block $\Longleftrightarrow$ **Multiprocessor**

Grid $\Longleftrightarrow$ **Device**

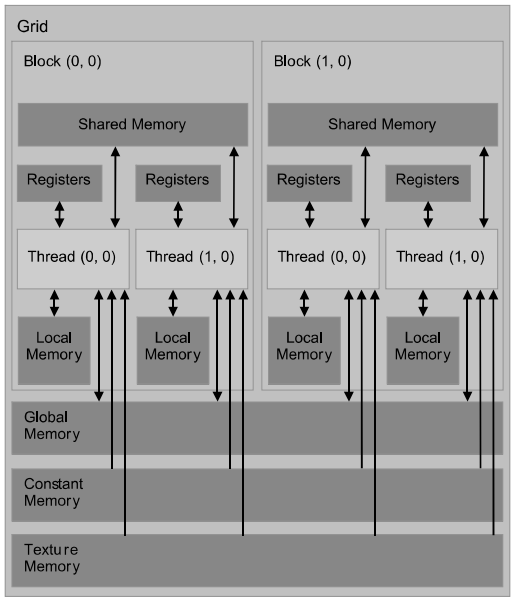
Scalability of a GPU kernel



Some capabilities of an Nvidia GPU

- An Nvidia GPU card can have 1 or more devices (Tesla S1070 = 4, Tesla C2070 = 1, NVS290 = 1, K20 = 1, K80 = 4, Titan X = 1)
- A single device can have several streamprocessors (SM) (NVS290 = 2, K20 = 13, K80 = 2x13, Titan X = 24, P100 = 28, V100=40)
- Each SM has 8 (1.x), 32 (2.0), 48 (2.x), 192 (3.x) or 128 (5/6.x) cores. A SM can run 32 active (semi) concurrent threads (called warps)
- Threads in a SM run concurrently (max. 8-32) or sequentially if more. Maximum resident threads per SM is 768 or 1536 (2.x and up).
- A block is a set of threads running on a single SM. Maximum number of threads per block is 512-1024 (x and y) and 64 (z). The total maximum (x*y*z) is 512 or 1024 (2.x and up)
- Maximum number of blocks is $2^{31} - 1$ for each dimension (for arch 2.x $2^{16} - 1$) independent of SM count. Arch 2.x and up have 3 dimensions (so x, y and z)

Memory organisation of a GPU 1



Memory organisation of a GPU 2

- Device memory consists of global, constant and texture memory
- Global memory accessible by all streaming multiprocessors and is persistent
- Constant and texture memory can only be *read* and have a **local cache** on each SM
- Each SM has shared memory (on-chip) that can be used by all threads in an active block but is non-persistent (kind of cache)
- Each SM has local memory (off-chip) and registers (on-chip) that can only be used by a single thread and which is also non-persistent
- Note that memory access times also depend on the access method ! Coalesced access may give a large performance boost, especially for less computation intensive problems

Unified Memory

A useful addition since architecture 3.0 is Unified Memory. This creates a **single virtual memory space** for all GPU(s) and host(s) memory and **automatically** copies data if accessed from a place where it is not physically located. As such it makes explicit `cudaMemcpy()` calls unnecessary.

Points of attention:

- Key word is `CudaMallocManaged()`
- Can sometimes be slower than manual memory management ...
- Must be compiled with architecture `sm_30` or higher
- Some Intel Memory Management Units may confuse this function, causing wrong answers without any warning!
  Fix: Use boot param `iommu=soft` on Linux. Windows: no idea :)

Example: Discuss and run `hello_unifiedmemory`

Parallel "Hello World2" in Cuda C with unified memory

```
#include <stdio.h>
#include <cuda.h>
#define NRBLKS   4 // Nr of blocks in a kernel (gridDim)
#define NRTPBK   4 // Nr of threads in a block (blockDim)

__global__ void decodeOnGPU(char *string)
{ int myid = (blockIdx.x * blockDim.x) + threadIdx.x;
  string[myid] ^= (char)011; // 000001001 = 011 in octal
}
int main(void)
{  char *encryption = "Aleef)^f{em)((\11";
   char * string; // pointer to unified memory
   int len = strlen(encryption);

   cudaMallocManaged((void **)&string, len);
   strncpy(string, encryption, len);
   decodeOnGPU <<< NRBLKS, NRTPBK >>> (string_d);

   cudaDeviceSynchronize();
   printf("%s\n",string);
   cudaFree(string);
}
```

Generic Cuda Blas framework Sgemv (unified memory)

```
#include <cuda.h>
#include <cublas_v2.h>
int main(void)
{
   float *h_A, *h_x, *h_b;  // matrix A and vectors b,x in managed memory
   float alpha = 1, beta = 0;
   cublasHandle_t handle;

   cublasCreate(&handle);
   cudaMallocManaged(&h_A, dim * dim * sizeof(float));
   cudaMallocManaged(&h_x, dim *       sizeof(float)); // and once more for b

   status = cublasSgemv(handle, ... , &alpha, h_A, dim, h_x, 1, &beta, h_b, 1);
   cudaDeviceSynchronize();  // must wait until result can be used !

   cublasDestroy(handle);
   cudaFree(h_A); cudaFree(h_x); cudaFree(h_b);
}
```


Debugging of memory leak errors using cuda-memcheck

cuda-memcheck is a memory leak checker for CUDA and works much the same as the Unix tool valgrind.

Notes

- Start it with : % `cuda-memcheck <PRG>`
- Cuda-memcheck always stops after the first memory error. You can force it to continue for the rest of the program using :
  % `cuda-memcheck --destroy-on-device-error kernel <PRG>` (5.x and up)

Example: Run the Memcheck-GPU example using memcheck and explain what is wrong in the code there.

Profiling Cuda code

Since CUDA version 5.x there is also a nice commandline profiler nvprof.

How to use: `nvprof myprogram <arguments>`
Kernel details: `nvprof --print-gpu-trace myprogram <arguments>`
Output for nvvp: `nvprof -o myprof.nvvp myprogram <arguments>`

There is also a GUI profiler named nvvp since Cuda 4.1 with more options but nvprof is really fine for basic profiling.

Example: Small demo for `matrixprod-simple` with 2 kernels in nvprof.

Example: Also show `parbody-psm1` (1024grid.par) with the nvvp GUI.

Exercise: Try to run nvprof and possibly nvvp by yourself.

Note: for nvvp set the executable name in File $->$ New Session $->$ File and add optional arguments before starting an analysis.

Some nice examples

Ok, so far the hard part of this Cuda course !

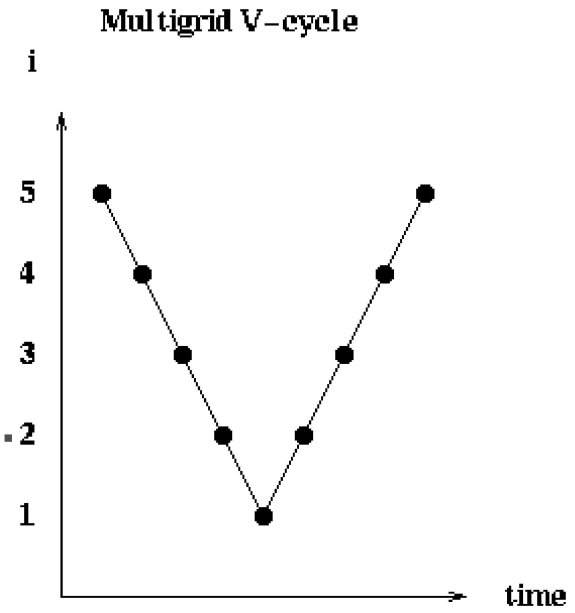
Let's try a few nice (graphic) examples, such as:

- fluidsGL
- smokeParticles
- randomFog
- particles
- Mandelbrot
- bandwidthTest

Multigrid Operators (2)

- The restriction operator R_i maps $P^{(i)}$ to $P^{(i-1)}$
  - Restricts problem on fine grid $P^{(i)}$ to coarse grid $P^{(i-1)}$ by sampling or averaging
  - $b_{i-1} = R_i (b_i)$
- The interpolation operator In_{i-1} maps an approximate solution x_{i-1} to an x_i
  - Interpolates solution on coarse grid $P^{(i-1)}$ to fine grid $P^{(i)}$
  - $x_i = In_{i-1} (x_{i-1})$
- The solution operator S_i takes $P^{(i)}$ and computes an improved solution x(i) on same grid
  - Uses "weighted" Jacobi or SOR
  - $x_{i, improved} = S_i (b_i , x_i)$
- Details of these operators follow after describing overall algorithm

Multigrid V-Cycle Algorithm (recursive)



Function MGV (b_i , x_i)

… Solve T_i $x_i = b_i$ given b_i and an initial guess for x_i

… return an improved x_i

if (i = 1)

compute exact solution x_1 of $P^{(1)}$ only 1 unknown.

return x_1

else

$x_i = S_i (b_i , x_i)$ improve solution by damping high frequency error,

$r_i = T_i\ x_i - b_i$ compute residual,

$d_i = In_{i-1}(MGV (R_i(r_i) , 0))$ solve $T_i\ d_i = r_i$ recursively,

$x_i = x_i - d_i$ correct fine grid solution,

$x_i = S_i(b_i, x_i)$ improve solution again.

return x_i

Complexity of a V-Cycle

° On a serial machine

- Work at each "dot" in the V-cycle is O(the number of unknowns)
- Cost of level i is $(2^i\text{-}1)^2 = O(4^i)$ (for a 2D grid)
- If finest grid level is m, total time is:

$$\sum_{i=1}^{m} O(4^i) = O(4^m) = O(\#\text{ unknowns})$$

° On a parallel machine (PRAM)

- with one processor per grid point and free communication, each step in the V-cycle takes constant time, O(1)
- Total V-cycle time is $O(m) = O(\log \#\text{unknowns})$

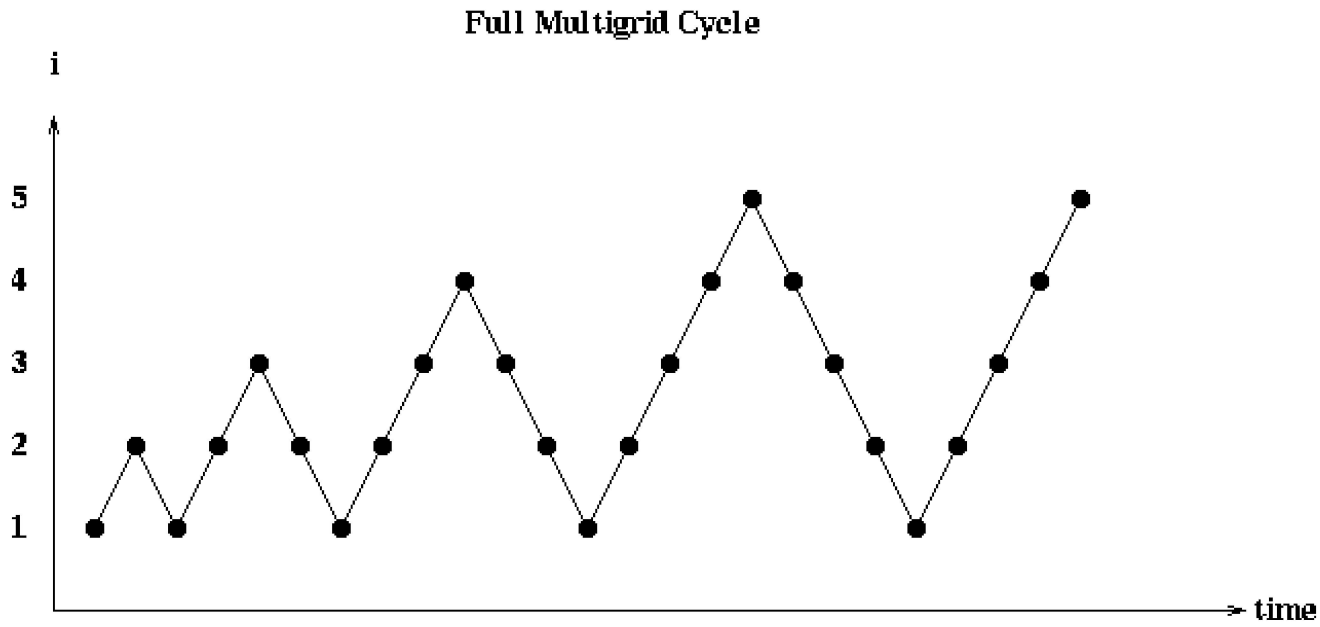
Full Multigrid (FMG)

- Intuition:
  - improve solution by doing multiple V-cycles
  - avoid expensive fine-grid (high frequency) cycles
  - analysis of why this works is beyond the scope of this class

```
Function FMG (b_m , x_m)
    … return improved x_m  given initial guess
    compute the exact solution x_1 of P^(1)
    for i=2 to m
        x_i = MGV ( b_i,  In_{i-1} (x_{i-1}) )
```

- In words:
  - Solve the problem with 1 unknown
  - Given a solution to the coarser problem, $P^{(i-1)}$, map it to starting guess for $P^{(i)}$
  - Solve the finer problem using the Multigrid V-cycle

Full Multigrid Cost Analysis



° One V for each call to FMG

- one also use "W"s and other compositions

° Serial time: $\Sigma_{i=1}^{m} O(4^i) = O(4^m) = O(\#\text{ unknowns})$

° PRAM time: $\Sigma_{i=1}^{m} O(i) = O(m^2) = O(\log^2 \#\text{ unknowns})$

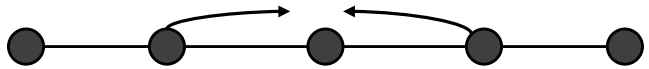
Complexity of Solving Poisson's Equation

- **Theorem: error ε after one FMG call is ≤ c times the error before, where c < 1/2, and independent of # unknowns**
- **x^k = FMG (b, x^{k-1}) ==> $\varepsilon(x^k) < 1/2\ \varepsilon(x^{k-1})$ (i.e., at least 1 bit per FMG iteration. x^k =solution after k^{th} FMG iteration)**
- **Corollary: We can make the error ε < tol, for any fixed tolerance in a fixed number of steps, independent of size of the finest grid**
- **This is the most important convergence property of MG, distinguishing it from other methods, which converge more slowly for large grids**
- **Total complexity is just proportional to the cost of one FMG call**

The Solution Operator S_i - Details

° The solution operator S_i , is a weighted Jacobi op.

° Consider the 1D problem



° At level i, pure Jacobi replaces:

$$x(j) := 1/2\ (x(j-1) + x(j+1) + b(j))$$

° Weighted Jacobi uses:

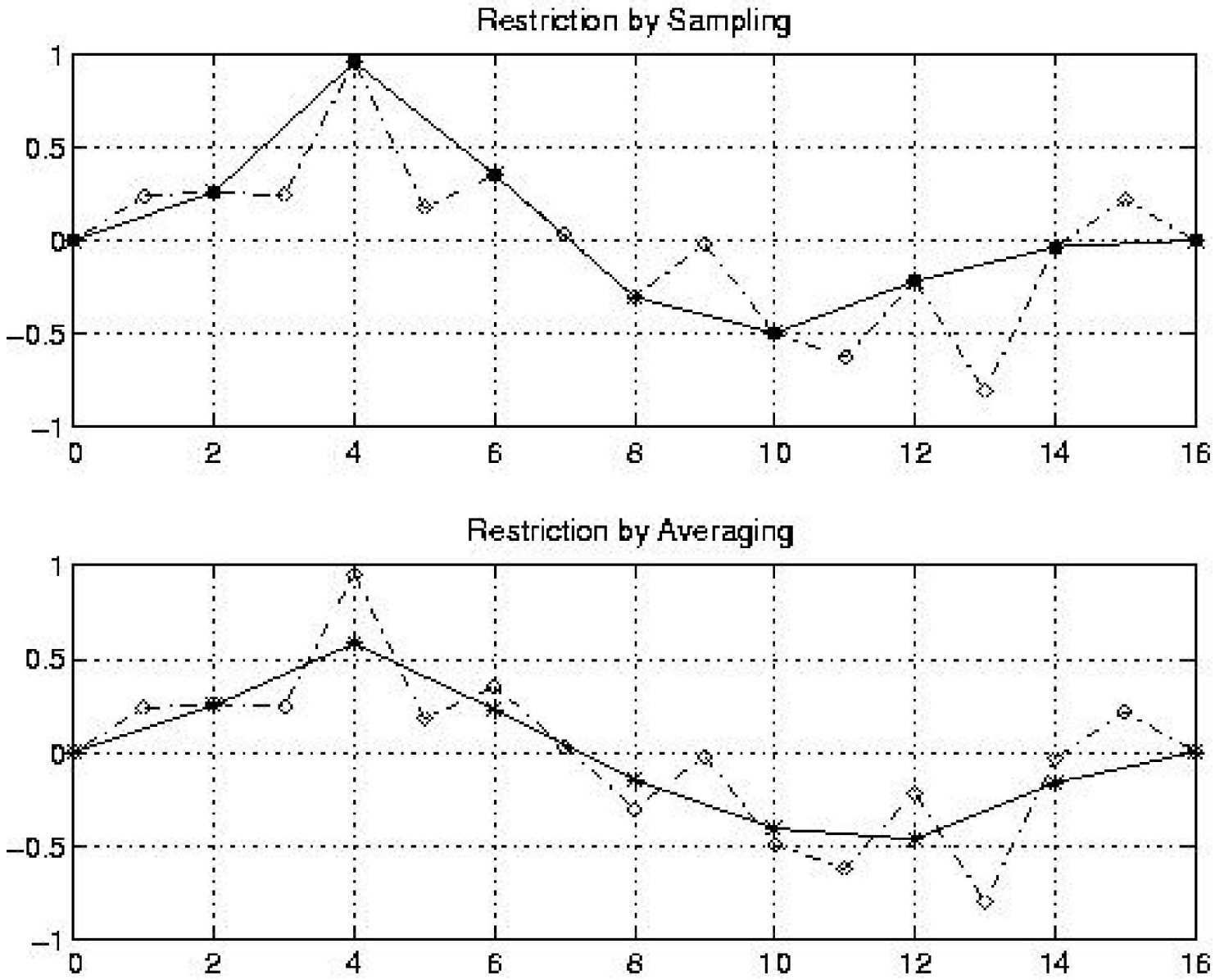
$$x(j) := 1/3\ (x(j-1) + x(j) + x(j+1) + b(j))$$

° In 2D, similar average of nearest neighbors

The Restriction Operator R_i - Details

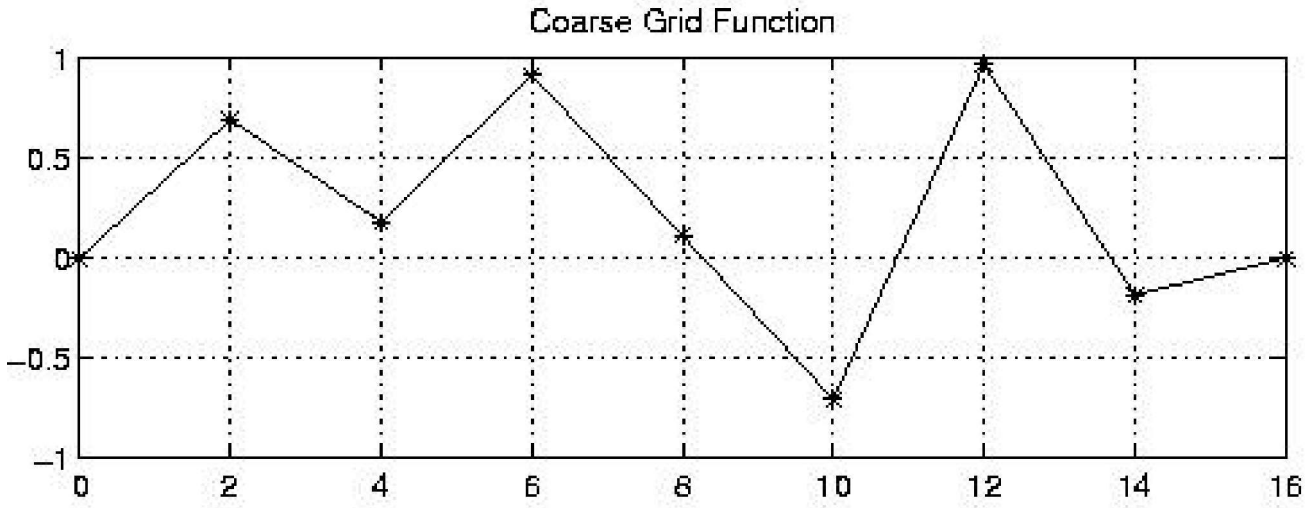
- The restriction operator R_i takes
  - a problem $P^{(i)}$ with RHS b_i and
  - maps it to a coarser problem $P^{(i-1)}$ with RHS b_{i-1}
- In 1D, average values of neighbors
  - $x_{coarse}(j) = 1/4 * x_{fine}(j-1) + 1/2 * x_{fine}(j) + 1/4 * x_{fine}(j+1)$

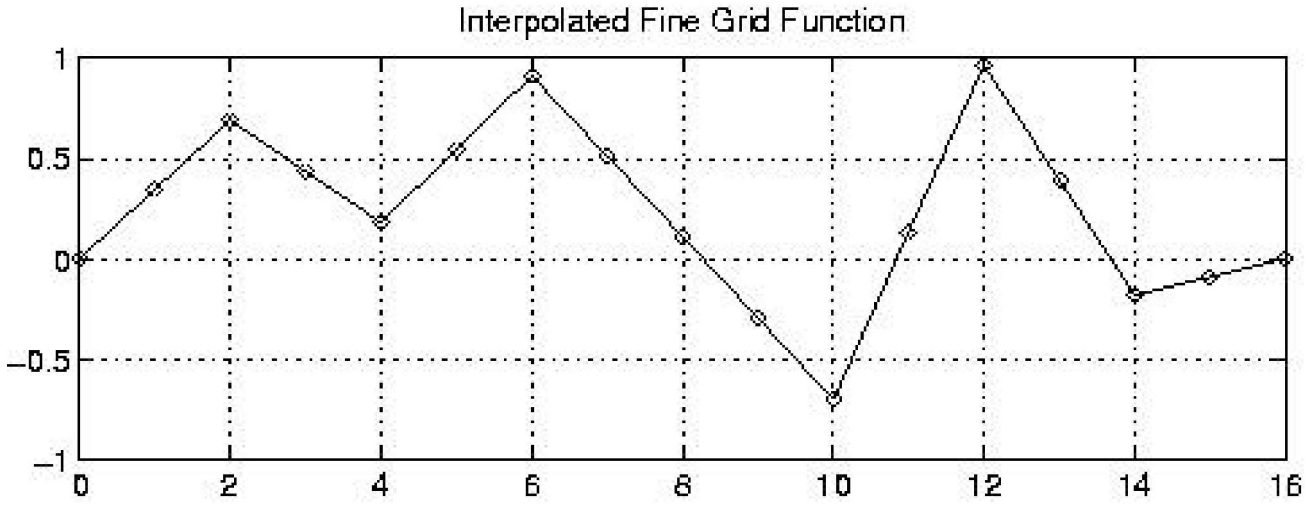
The Restriction Operator R_i - Details



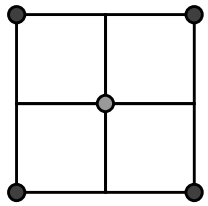
° **In 2D, average with all 8 neighbors (N,S,E,W,NE,NW,SE,SW)**

Interpolation Operator In_i





° **In 2D, interpolation requires averaging with 4 nearest neighbors (NW,SW,NE,SE)**



Performance Model of parallel 2D Multigrid

- Assume 2^m+1 by 2^m+1 grid of unknowns, $n= 2^m+1$, $N=n^2$
- Assume $p = 4^k$ processors, arranged in 2^k by 2^k grid
  - Each processor starts with 2^{m-k} by 2^{m-k} subgrid of unknowns
- Consider V-cycle starting at level m
  - At levels m through k of V-cycle, each processor does some work
  - At levels k-1 through 1, some processors are idle, because a 2^{k-1} by 2^{k-1} grid of unknowns cannot occupy each processor

Performance Model of parallel 2D Multigrid (2)

° **Cost of one level (j) or $P^{(j)}$ in V-cycle**

- **If level j >= k, then cost =**

  $O(4^{j-k})$ …. **Flops, proportional to the number of grid points/processor**

  + O(1) α …. **Send a constant # messages to neighbors**

  + $O(2^{j-k})$ β …. **Number of words sent**

- **If level j < k, then cost =**

  O(1) …. **Flops, proportional to the number of grid points/processor**

  + O(1) α …. **Send a constant # messages to neighbors**

  + O(1) β …. **Number of words sent**

° **Sum over all levels in all V-cycles in FMG to get complexity**

Comparison of Methods (using p processors)

	# Flops	# Messages	# Words sent
MG	$N/p + \log p * \log N$	$(\log N)^2$	$(N/p)^{1/2} + \log p * \log N$
FFT	$N \log N / p$	$p^{1/2}$	N/p
SOR	$N^{3/2}/p$	$N^{1/2}$	N/p

- SOR is slower than others on all counts
- Flops for MG and FFT depends on accuracy of MG
- MG communicates less total data (bandwidth)
- Total messages (latency) depends …
  - This coarse analysis can't say whether MG or FFT is better when $\alpha >> \beta$

Practicalities

- In practice, we don't go all the way to $P^{(1)}$
- In sequential code, the coarsest grids are negligibly cheap, but on a parallel machine they are not.
  - Consider 1000 points per processor
  - In 2D, the surface to communicate is 4*sqrt(1000) ~= 128, or 13%
  - In 3D, the surface is $1000\text{-}8^3$ ~= 500, or 50%
  - Data locality ratio α (large α is preferred):

$$\alpha = \frac{\text{computation time between two communication steps}}{\text{communication time}}$$

- Apply communication avoiding Jacobi iterations!
- Dealing with coarse meshes efficiently
  - Should we switch to using fewer processors on coarse meshes?
  - Should we switch to another solver on coarse meshes?

Performance metrics

A pragmatic classification:

- Users :
  - How fast is my application?
    - *execution time*
- Developers:
  - How close to the absolute best can I be?
    - *estimate absolute best*
    - *compute performance gain (speed-up)*
- Budget-holders:
  - How much of my infrastructure am I using?
    - *efficiency*
    - *utilization*

Performance "actions"

- Performance measurement
  - *measure execution time*
  - *derive metrics such as speed-up, throughput, bandwidth*
  - *platform and application implementation are available*
  - *data-sets are available*
- Performance analysis
  - estimate performance bounds
    - *performance bounds are typically worst-case, best-case, average-case scenarios*
  - platform and application are available/models
  - data-sets are available/models
- Performance prediction
  - estimate application behavior
  - platform and application are models
  - data-sets are real

Performance Metrics

- Serial execution time: T_S
- Parallel execution time: T_P
- Overhead (p is # compute units) $T_O = p \cdot T_P - T_S$
  - *ideal case: $T_o = 0$ (perfect linear speed-up)*

Total overhead relevant for dedicated parallel processing

- Speedup $S = \frac{T_{serial\_best}}{T_P}$

Overhead may **depend** on p!

Relative versus true speedup: using $T_P(P=1)$ instead of T_{serial_best}

Superlinear speed-up is sometimes possible:
cache effects / memory sizes

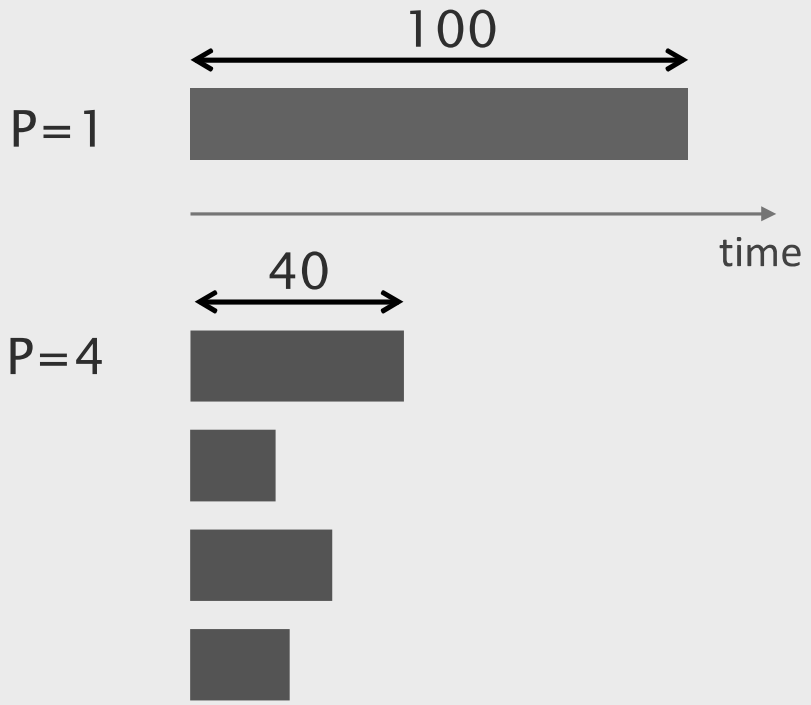
Efficiency

$$E = \frac{S}{p}$$

$$E = \frac{S}{p} = \frac{T_S}{p \cdot T_p}$$

$$T_O = p \cdot T_P - T_S$$

$$E = \frac{1}{1 + \frac{T_o}{T_S}}$$



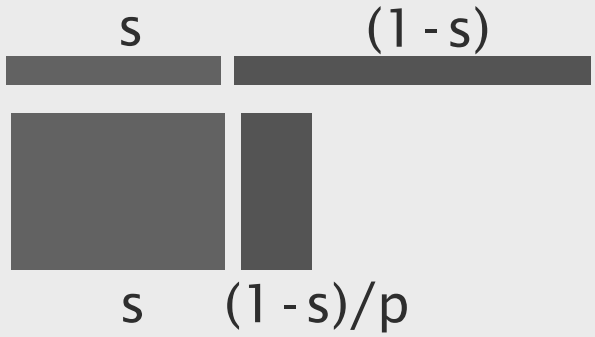
S=2.5
E=2.5/4=0.625

Sources of overhead in parallel programs

- Inter-process interaction
  - *communication => idling*
  - *synchronization => serialization*
- Load imbalance
  - *un-even workloads => idling*
- Additional computations, such as
  - *memory allocation*
  - *data partitioning*
  - *managing the parallelism*
  - *…*

Recap: Amdahl's law – fixed problem size

- Every application has an intrinsically sequential part
- Amdahl's law:
  - *let* s *be the fraction of work that is sequential, then (1-s) is the fraction that is parallelizable*
  - *p = number of processors*
  - *S = Speedup*



$$S = T_{seq}/T_{par} = 1/(s + (1-s)/p) \leq 1/s$$

Speedup is bounded by the sequential fraction.

Isoefficiency function

What is a 'good' efficiency of a parallel program? →No simple answer

Emphasis on scalablity of a parallel algorithm: can a program retain its efficiency when #processors and problem size increase?

$T_o = p \cdot T_P - W$ (definition of overhead)

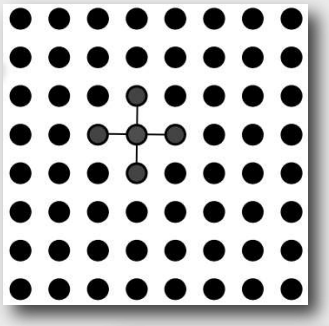
and

$$S = \frac{W}{T_P} \Rightarrow S = \frac{W \cdot p}{W + T_o} \Rightarrow E = \frac{1}{1 + T_o/W}$$

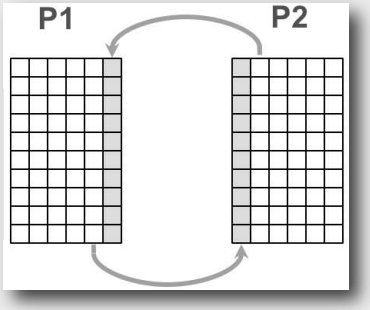
Conclusions:

1. E=1 when there is no overhead
2. E is fixed iff T_o/W is fixed

Example: stencil-type computations (1/3)



(n+2)x(n+2) grid
given boundary values



column-wise
data distribution

```
a[i,j]=0.25*(a[i,j-1]+a[i-1,j]+a[i,j+1]+a[i+1,j]);
```

$$T_S = 4t_{op} \cdot n^2$$

$$T_{comm} = 2n \cdot t_{data}$$
$$T_{calc} = 4t_{op} \cdot (n \cdot n/p) = 4t_{op}n^2/p$$

$$T_P = 4t_{op} \cdot n^2/p + 2n \cdot t_{data}$$

Example: stencil-type computations (2/3)

$$S = \frac{t_{op} n^2 \cdot p}{t_{op} n^2 + p \cdot n \cdot t_{data}/2}$$

$$E = \frac{S}{p} = \frac{1}{1 + (p/n)(t_{data}/2t_{op})}$$

- So *p* must be small relative to *n* for efficiency
- Efficiency stays constant as long as p/n is constant

Hardware Performance metrics

- Clock frequency [GHz] = absolute hardware speed
  - *memories, CPUs, interconnects*
- Operational speed [GFLOPs]
  - *how many operations per cycle a machine can do*
- Memory bandwidth (BW) [GB/s]
  - *differs a lot between different memories on chip*
  - *remember? Slow memory is large, fast memory is small …*
- Power [Watt]
- Derived metrics
  - *normalized for comparison purposes …*
  - *FLOPs/Byte, FLOPs/Watt, …*

Theoretical peak performance

Peak = chips * cores * vectorWidth *
FLOPs/cycle * clockFrequency

- *cores = real cores, hardware threads, or ALUs, depending on the architecture*

Examples from DAS-4:

- Intel Core i7 CPU

2 chips * 4 cores * 4-way vectors * 2 FLOPs/cycle * 2.4 GHz = **154 GFLOPs**

- NVIDIA GTX 580 GPU

1 chip * 16 SMs * 32 cores * 2 FLOPs/cycle * 1.544 GHz = **1581 GFLOPs**

- ATI AMD Radeon HD 6970 GPU

1 chip * 24 SIMD engines * 16 cores * 4-way vectors * 2 FLOPs/cycle * 0.880 GHz = **2703 GFLOPs**

DRAM Memory bandwidth (off-chip)

Throughput = memory bus frequency * bits per cycle * bus width

- *memory clock is not the CPU clock (typically lower)*
- *divide by 8 to get B/s*

Examples:

- Intel Core i7 DDR3: 1.333 GHz * 2 * 64 = **21 GB/s**
- NVIDIA GTX 580 GDDR5: 1.002 GHz * 4 * 384 = **192 GB/s**
- ATI HD 6970 GDDR5: 1.375 GHz * 4 * 256 = **176 GB/s**

Power

- Chip manufacturers specify Thermal Design Power (TDP)
  - *some definition of maximum power consumption …*
- We can measure dissipated power
  - *whole system*
  - *typically (much) lower than TDP*
- Power efficiency: FLOPs / Watt
- Examples (with theoretical peak and TDP)
  - Intel Core i7: 154 / 160 = **1.0 GFLOPs/W**
  - NVIDIA GTX 580: 1581 / 244 = **6.3 GFLOPs/W**
  - ATI HD 6970: 2703 / 250 = **10.8 GFLOPs/W**

Software metrics (3 P's)

Performance metrics

- Execution time
  - *Derive speed-up vs. best available sequential performance*
- Achieved GFLOPs:
  - *Count (FL)OPs, divide by execution time => FLOPS/s*
  - *Derive computational efficiency (i.e., utilization) =* $\frac{Achieved\ FLOPs}{Peak\ FLOPs}$
- Achieved GB/s:
  - *Count memory OPs, divide by execution time => B/s*
  - *Derive memory efficiency (i.e., utilization) =* $\frac{Achieved\ GB/s}{Peak\ GB/s}$

Productivity and ***P***ortability metrics

- Programmability
- Production costs
- Maintenance costs

Attainable performance

- Attainable GFlops/sec

  = *min (* Peak Floating-Point Performance,

  Peak Memory Bandwidth * Operational Intensity)

- To translate:
  - if an application is compute-bound =>

    performance is limited by peak performance
  - if an application is memory-bound =>

    performance is limited by the load it puts on the memory system

Use the Roofline model

Determine what to do first to gain performance

- increase memory streaming rate (fights mem-boundness)
  - *GPU: memory coalescing*
  - *CPUs: better caching*
- apply in-core optimizations (fights compute-boundness)
  - *vectorization*
- increase arithmetic intensity (fights mem-boundness)
  - *change your algorithm*
  - *think of new ways to reuse the data*

Nodal Coordinates: Inertial Partitioning

- For a graph in 2D, choose line with half the nodes on one side and half on the other
  - In 3D, choose a plane, but consider 2D for simplicity
- Determine a line L, and then choose a line $L^{\perp}$ perpendicular to it, with half the nodes on either side

1. **Determine a line L through the points**

   L given by a*(x-xbar)+b*(y-ybar)=0,

   with $a^2+b^2=1$; (a,b) is unit vector $\perp$ to L

2. **Project each point to the line**

   For each $n_j = (x_j, y_j)$, compute coordinate

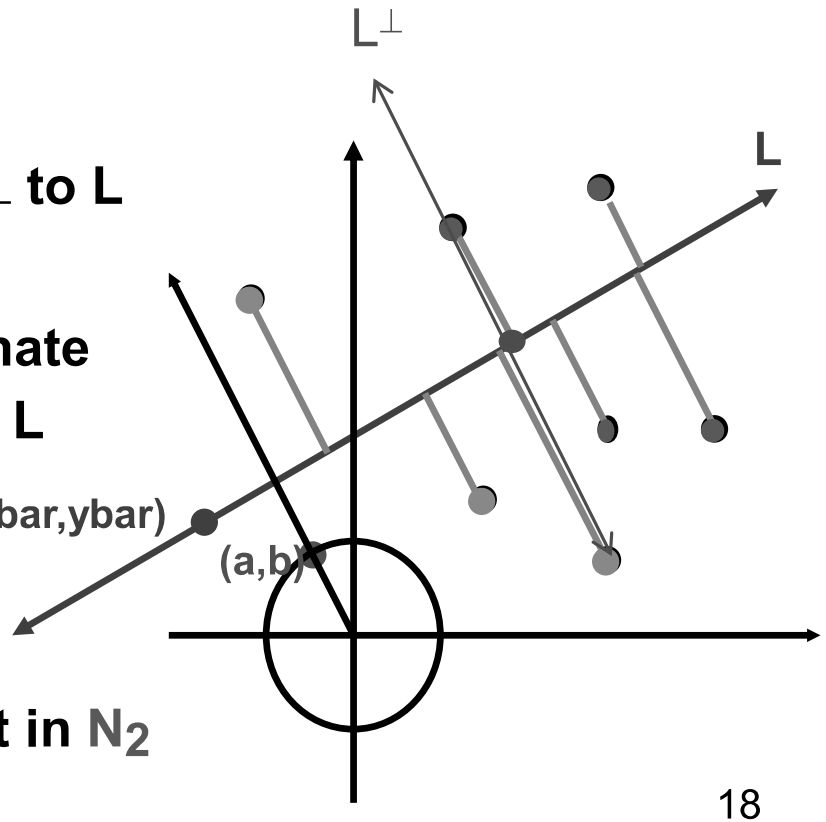
   $S_j = -b*(x_j-xbar) + a*(y_j-ybar)$ along L

3. **Compute the median**

   Let Sbar = median(S_1,…,S_n)

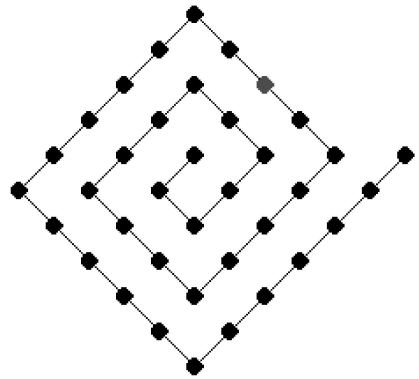
4. **Use median to partition the nodes**

   Let nodes with S_j < Sbar be in N_1, rest in N_2



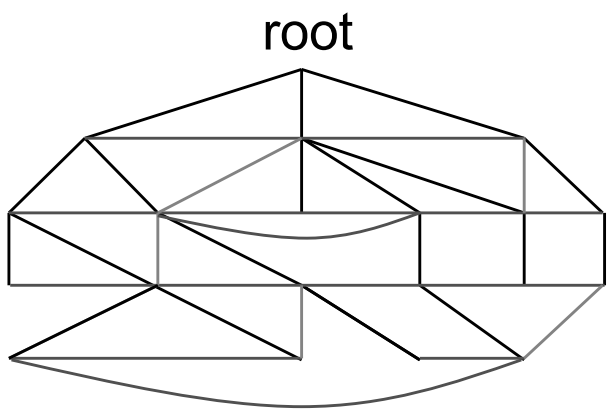
Nodal Coordinates: Summary

- Other variations on these algorithms
- Algorithms are efficient (i.e., fast)
- Rely on graphs having nodes connected (mostly) to "nearest neighbors" in space
  - algorithm does not depend on where actual edges are!
- Common when graph arises from physical model
- Ignores edges, but can be used as good starting guess for subsequent partitioning that does examine edges
- Can do poorly if graph connection is not spatial:



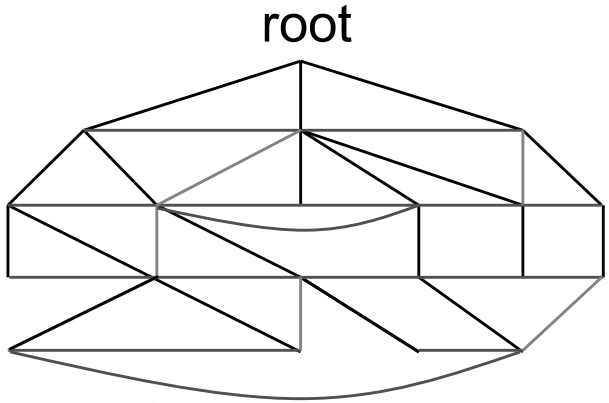
Breadth First Search (details)

- Queue (First In First Out, or FIFO)
  - Enqueue(x,Q) adds x to back of Q
  - x = Dequeue(Q) removes x from front of Q
- Compute Tree $T(N_T,E_T)$

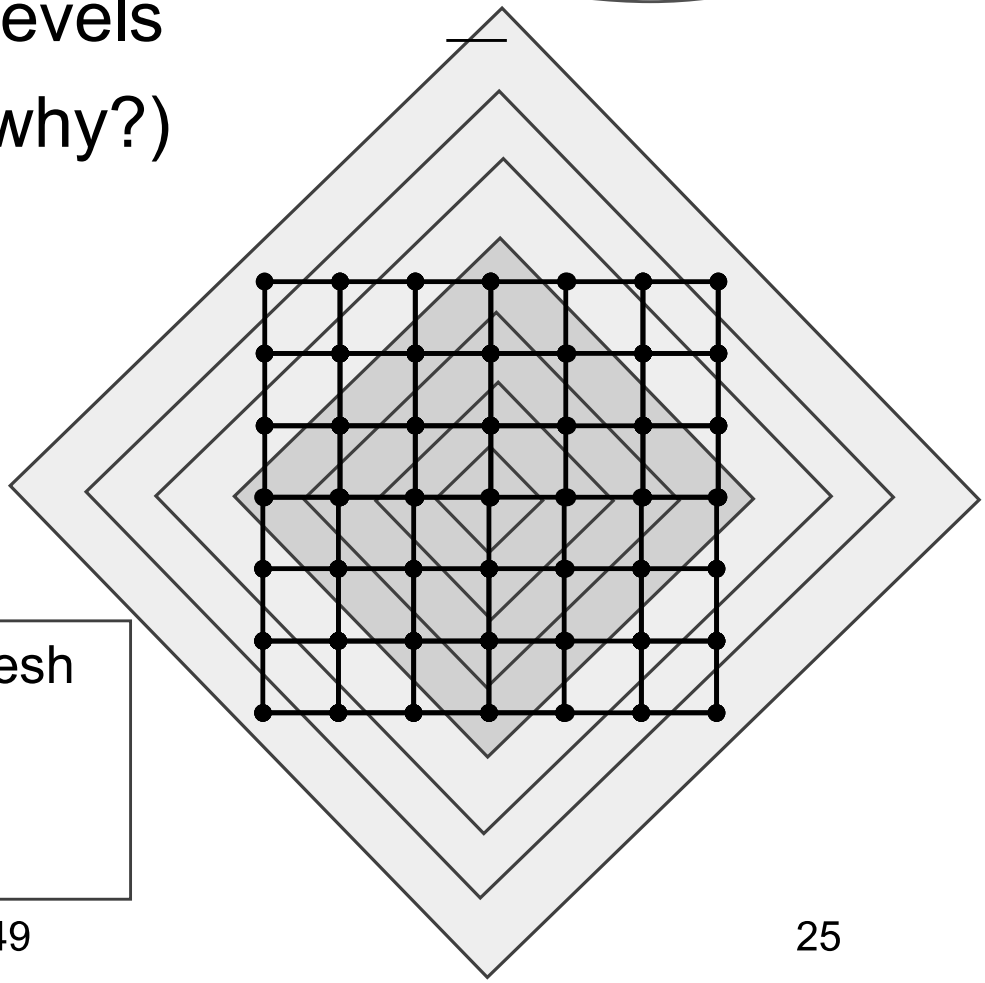


N_T = {(r,0)}, E_T = empty set … Initially T = root r, which is at level 0
Enqueue((r,0),Q) … Put root on initially empty Queue Q
Mark r … Mark root as having been processed
While Q not empty … While nodes remain to be processed
    (n,level) = Dequeue(Q) … Get a node to process
    For all unmarked children c of n
        $N_T = N_T$ U (c,level+1) … Add child c to N_T
        $E_T = E_T$ U (n,c) … Add edge (n,c) to E_T
        Enqueue((c,level+1),Q)) … Add child c to Q for processing
        Mark c … Mark c as processed
    Endfor
Endwhile

Partitioning via Breadth First Search



- BFS identifies 3 kinds of edges
  - Tree Edges - part of T
  - Horizontal Edges - connect nodes at same level
  - Interlevel Edges - connect nodes at adjacent levels
- No edges connect nodes in levels differing by more than 1 (why?)
- BFS partioning heuristic
  - N = N_1 U N_2, where
    - N_1 = {nodes at level <= L},
    - N_2 = {nodes at level > L}
  - Choose L so $|N_1|$ close to $|N_2|$



BFS partition of a 2D Mesh using center as root:
N1 = levels 0, 1, 2, 3
N2 = levels 4, 5, 6

Kernighan/Lin: Preliminary Definitions

- T = cost(A, B), newT = cost(newA, newB)
- Need an efficient formula for newT; will use
  - E(a) = external cost of a in A = Σ {W(a,b) for b in B}
  - I(a) = internal cost of a in A = Σ {W(a,a') for other a' in A}
  - D(a) = cost of a in A = E(a) - I(a)
  - E(b), I(b) and D(b) defined analogously for b in B
- Consider swapping X = {a} and Y = {b}
  - newA = (A - {a}) U {b}, newB = (B - {b}) U {a}
- newT = T - (D(a) + D(b) - 2*w(a,b)) ≡ T - gain(a,b)
  - gain(a,b) measures improvement gotten by swapping a and b
- Update formulas (cost changes only when (a',a) or (a',b) exist)
  - newD(a') = D(a') + 2*w(a',a) - 2*w(a',b) for a' in A, a' ≠ a
  - newD(b') = D(b') + 2*w(b',b) - 2*w(b',a) for b' in B, b' ≠ b

Kernighan/Lin Algorithm

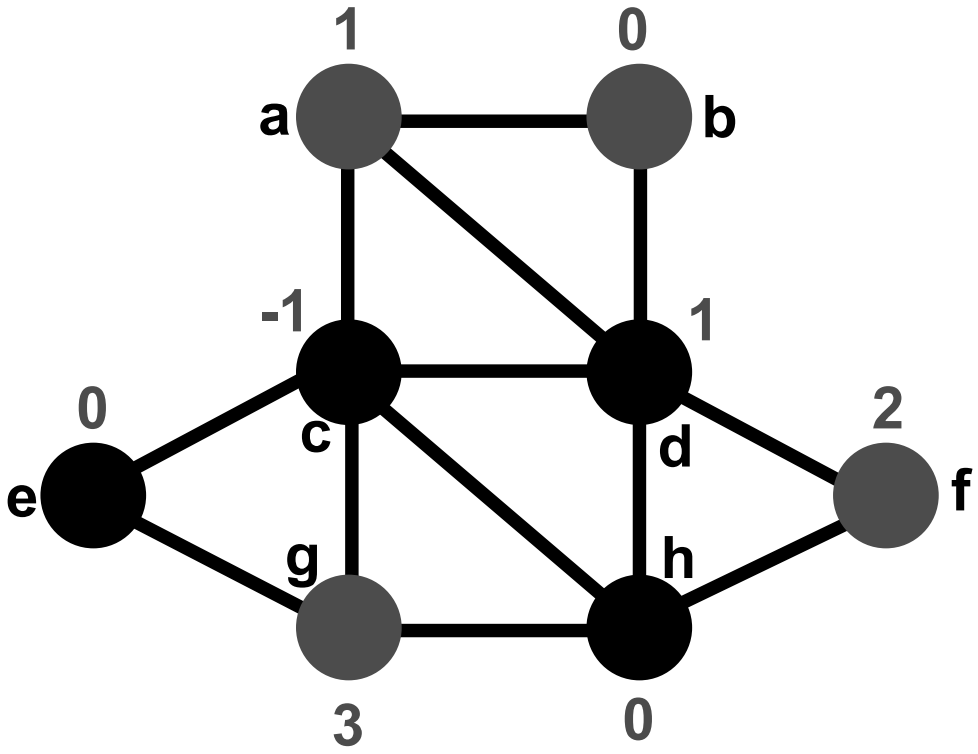
```
Compute  T = cost(A,B) for initial A, B                    … cost = O(|N|^2)
Repeat
    … One pass greedily computes |N|/2 possible X,Y to swap, picks best
    Compute costs D(n) for all n in N                      … cost = O(|N|^2)
    Unmark all nodes in N                                  … cost = O(|N|)
    While there are unmarked nodes                         … |N|/2 iterations
        Find an unmarked pair (a,b) maximizing gain(a,b)   … cost = O(|N|^2)
        Mark a and b (but do not swap them)                … cost = O(1)
        Update D(n) for all unmarked n,
            as though a and b had been swapped             … cost = O(|N|)
    Endwhile
        … At this point we have computed a sequence of pairs
        …  (a1,b1), … , (ak,bk)   and gains gain(1),…., gain(k)
        … where k = |N|/2, numbered in the order in which we marked them
    Pick m maximizing Gain = Σ_{k=1 to m} gain(k)          … cost = O(|N|)
        … Gain is reduction in cost from swapping (a1,b1) through (am,bm)
    If Gain > 0 then   … it is worth swapping
        Update newA = A - { a1,…,am } U { b1,…,bm }        … cost = O(|N|)
        Update newB = B - { b1,…,bm } U { a1,…,am }        … cost = O(|N|)
        Update T = T - Gain                                … cost = O(1)
    endif
Until Gain <= 0
```

Simplified Fiduccia-Mattheyses: Example (1)

Red nodes are in Part1; black nodes are in Part2.

The initial partition into two parts is arbitrary. In this case it cuts 8 edges.

The initial node gains by changing membership Part are shown in red.



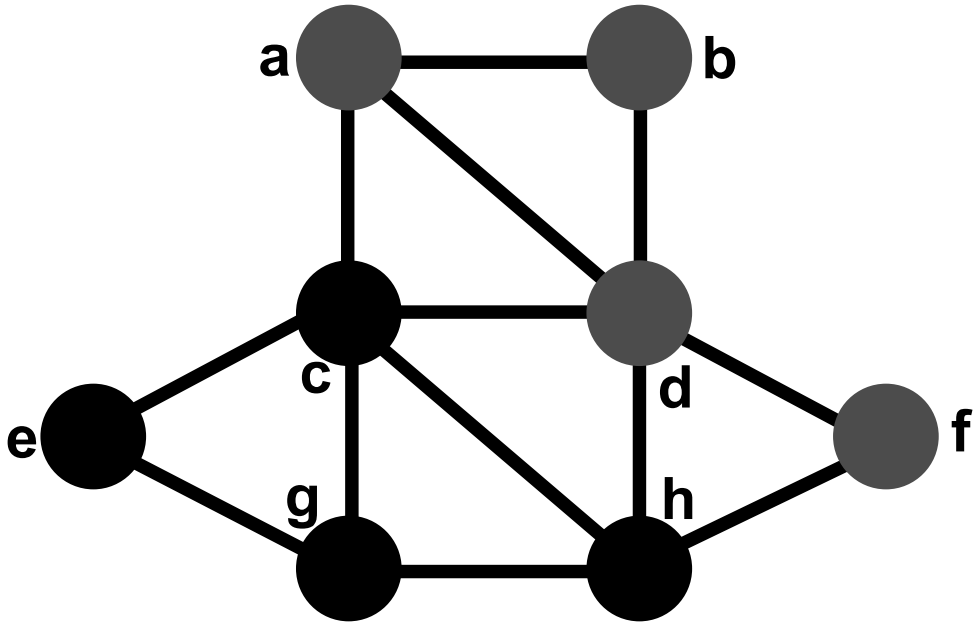
Nodes tentatively moved (and cut size after each pair):

none (8);

Simplified Fiduccia-Mattheyses: Example (10)

After every node has been tentatively moved, we look back at the sequence and see that the smallest cut was 4, after swapping g and d. We make that swap permanent and undo all the later tentative swaps.

This is the end of the first improvement step.



Nodes tentatively moved (and cut size after each pair):

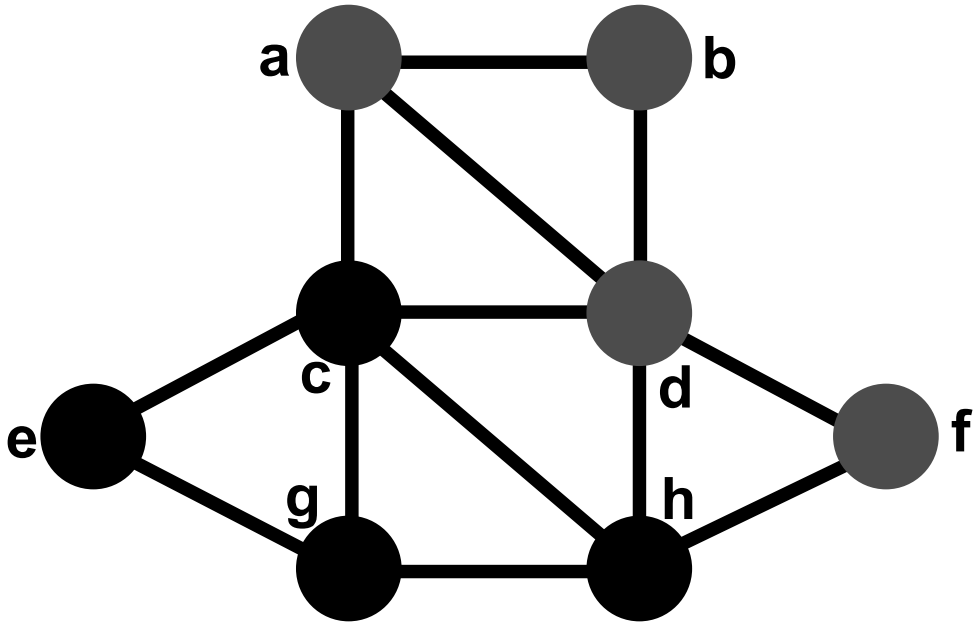
none (8); **g, d (4);** f, c (5); b, e (7); a, h (8)

Simplified Fiduccia-Mattheyses: Example (11)

Now we recompute the gains and do another improvement step starting from the new size-4 cut. The details are not shown.

The second improvement step doesn't change the cut size, so the algorithm ends with a cut of size 4.

In general, we keep doing improvement steps as long as the cut size keeps getting smaller.



Basic Definitions

- *Definition*: The incidence matrix In(G) of a graph G(N,E) is an |N| by |E| matrix, with one row for each node and one column for each edge. For an edge e=(i,j), column e of In(G) is zero except for the i-th and j-th entries, which are +1 and -1, respectively.
- Slightly ambiguous definition because multiplying column e of In(G) by -1 still satisfies the definition, but this won't matter...
- *Definition*: The Laplacian matrix L(G) of a graph G(N,E) is an |N| by |N| symmetric matrix, with one row and column for each node. It is defined by
  - L(G) (i,i) = degree of node i (number of incident edges)
  - L(G) (i,j) = -1 if i ≠ j and there is an edge (i,j)
  - L(G) (i,j) = 0 otherwise

Properties of Laplacian Matrix

- *Theorem 1:* Given G, L(G) has the following properties
  (proof on 1996 CS267 web page)
  - L(G) is symmetric.
    - This means the eigenvalues of L(G) are real and its eigenvectors are real and orthogonal.
  - $In(G) * (In(G))^T = L(G)$
  - The eigenvalues of L(G) are nonnegative:
    - $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$
  - The number of connected components of G is equal to the number of λ_i equal to 0.
  - *Definition*: $\lambda_2(L(G))$ is the algebraic connectivity of G
    - The magnitude of λ_2 measures connectivity
    - In particular, $\lambda_2 \neq 0$ if and only if G is connected.

Spectral Bisection Algorithm

- Spectral Bisection Algorithm:
  - Compute eigenvector v_2 corresponding to $\lambda_2(L(G))$
  - For each node n of G
    - if $v_2(n) < 0$ put node n in partition N-
    - else put node n in partition N+
- Why does this make sense? First reasons...
  - *Theorem 2 (Fiedler, 1975):* Let G be connected, and N- and N+ defined as above. Then N- is connected. If no $v_2(n) = 0$, then N+ is also connected.
  - Recall $\lambda_2(L(G))$ is the algebraic connectivity of G
  - *Theorem 3 (Fiedler):* Let $G_1(N,E_1)$ be a subgraph of G(N,E), so that G_1 is "less connected" than G. Then $\lambda_2(L(G_1)) \leq \lambda_2(L(G))$, i.e. the algebraic connectivity of G_1 is less than or equal to the algebraic connectivity of G.

Spectral Bisection Algorithm

- Spectral Bisection Algorithm:
  - Compute eigenvector v_2 corresponding to $\lambda_2(L(G))$
  - For each node n of G
    - if $v_2(n) < 0$ put node n in partition N-
    - else put node n in partition N+
- Why does this make sense? More reasons...
  - *Theorem 4 (Fiedler, 1975):* Let G be connected, and N1 and N2 be any partition into part of equal size |N|/2. Then the number of edges connecting N1 and N2 is at least .25 * |N| * $\lambda_2(L(G))$.

Computing v_2 and λ_2 of L(G) using Lanczos

- Given any n-by-n symmetric matrix A (such as L(G)) Lanczos computes a k-by-k "approximation" T by doing k matrix-vector products, k << n

```
Choose an arbitrary starting vector r
b(0) = ||r||
j=0
repeat
    j=j+1
    q(j) = r/b(j-1)          … scale a vector (BLAS1)
    r = A*q(j)               … matrix vector multiplication, the most expensive step
    r = r - b(j-1)*v(j-1)    … "axpy", or scalar*vector + vector (BLAS1)
   a(j) = v(j)^T * r         … dot product (BLAS1)
   r = r - a(j)*v(j)         … "axpy" (BLAS1)
   b(j) = ||r||              … compute vector norm (BLAS1)
until convergence            … details omitted
```

$$T = \begin{pmatrix} a(1) & b(1) & & & & \\ b(1) & a(2) & b(2) & & & \\ & b(2) & a(3) & b(3) & & \\ & & \ddots & \ddots & \ddots & \\ & & & b(k-2) & a(k-1) & b(k-1) \\ & & & & b(k-1) & a(k) \end{pmatrix}$$

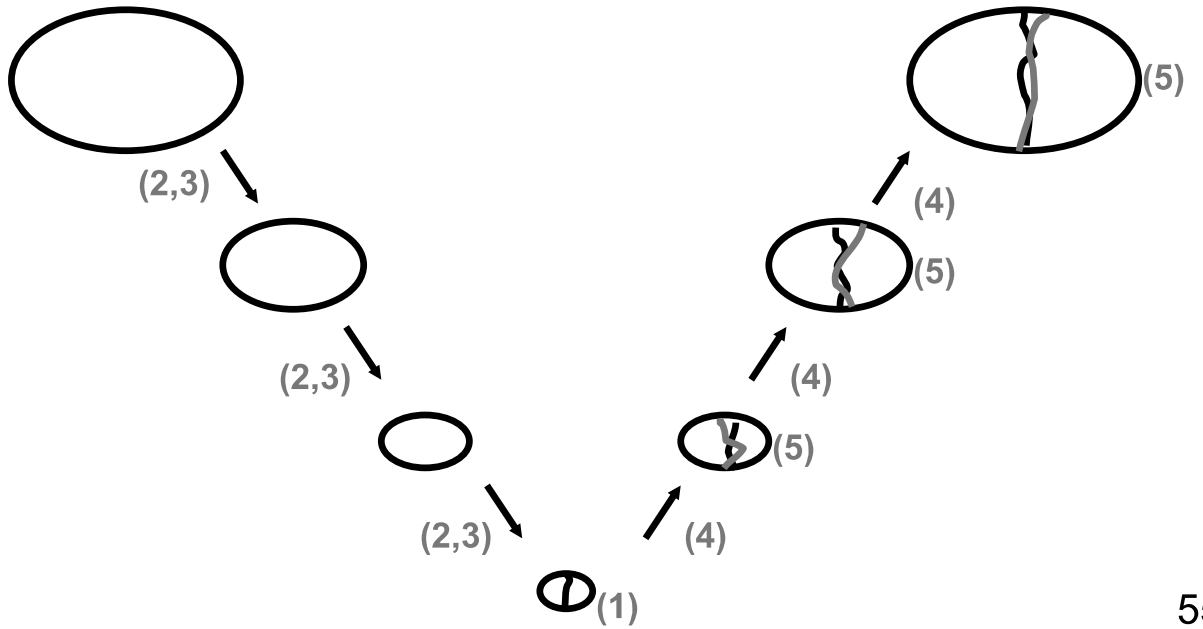
- Approximate A's eigenvalues/vectors using T's

Multilevel Partitioning - High Level Algorithm

```
(N+,N- ) = Multilevel_Partition( N, E )
        … recursive partitioning routine returns N+ and N- where N = N+ U N-
        if |N| is small
(1)         Partition G = (N,E)  directly to get N = N+ U N-
            Return (N+, N- )
        else
(2)         Coarsen G to get an approximation Gc = (Nc, Ec)
(3)         (Nc+ , Nc- ) = Multilevel_Partition( Nc, Ec )
(4)         Expand (Nc+ , Nc- ) to a partition  (N+ , N- ) of N
(5)         Improve the partition ( N+ , N- )
            Return ( N+ , N- )
        endif
```

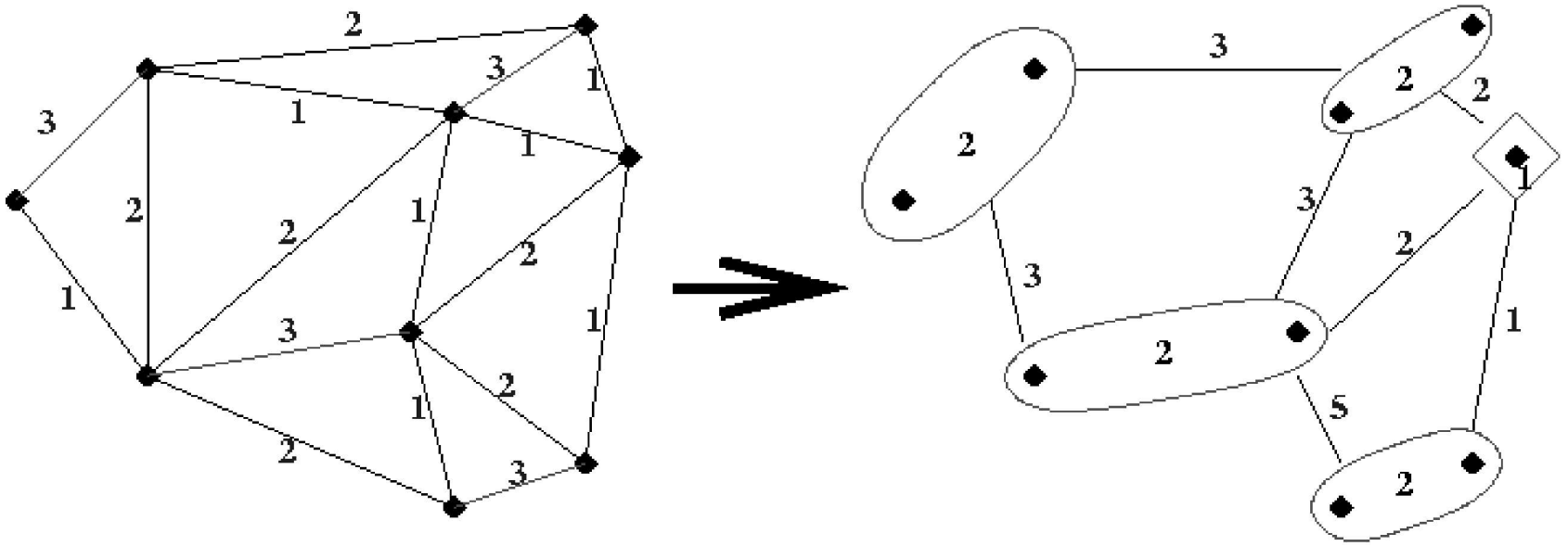
"V - cycle:"

How do we
- **Coarsen?**
- **Expand?**
- **Improve?**



Example of Coarsening

How to coarsen a graph using a maximal matching



$G = (N, E)$

E_m is shown in red

Edge weights shown in blue

Node weights are all one

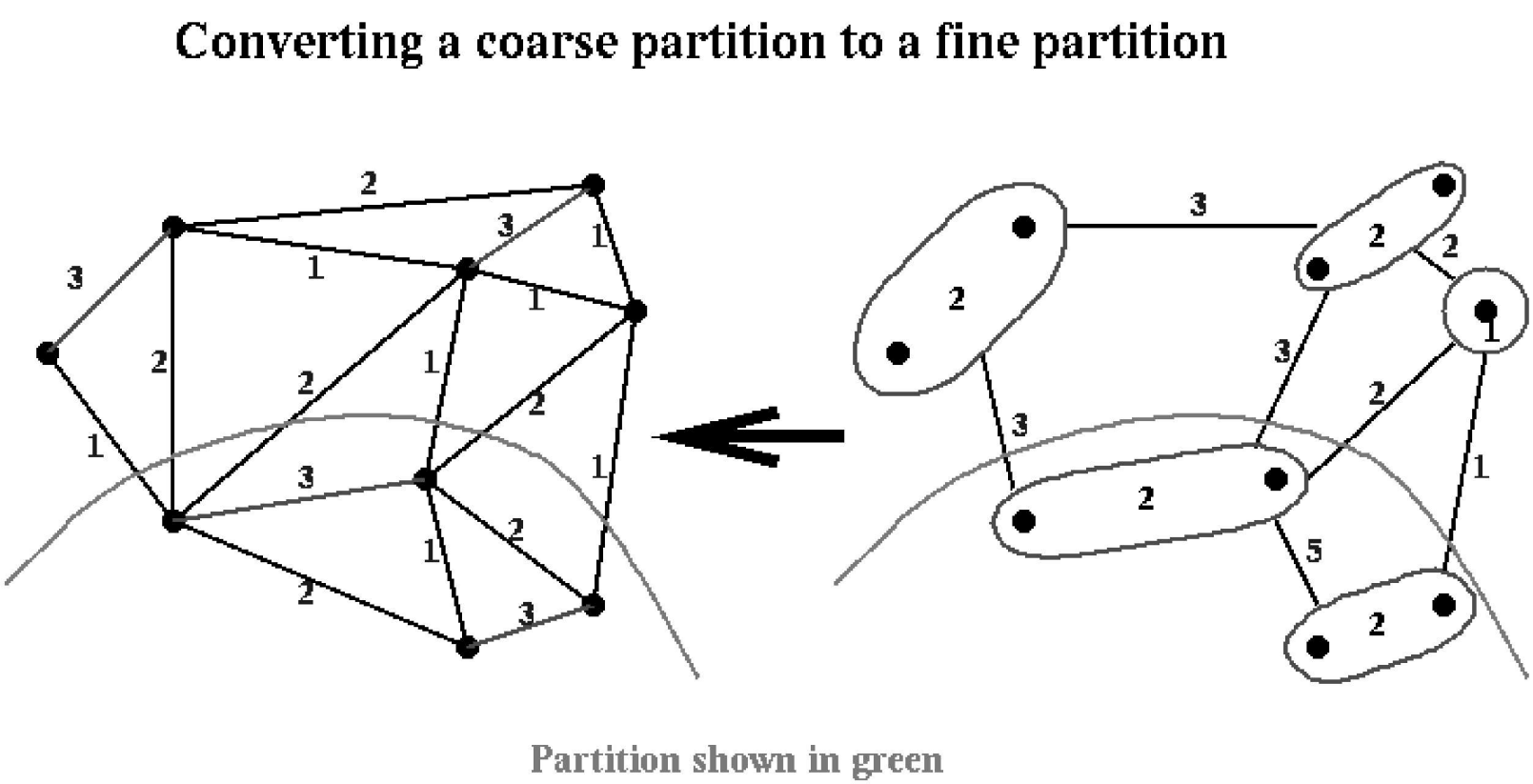
$G_c = (N_c, E_c)$

N_c is shown in red

Edge weights shown in blue

Node weights shown in black

Expanding a partition of G_c to a partition of G



Particle Simulation, bottlenecks

° **Computational bottlenecks?**

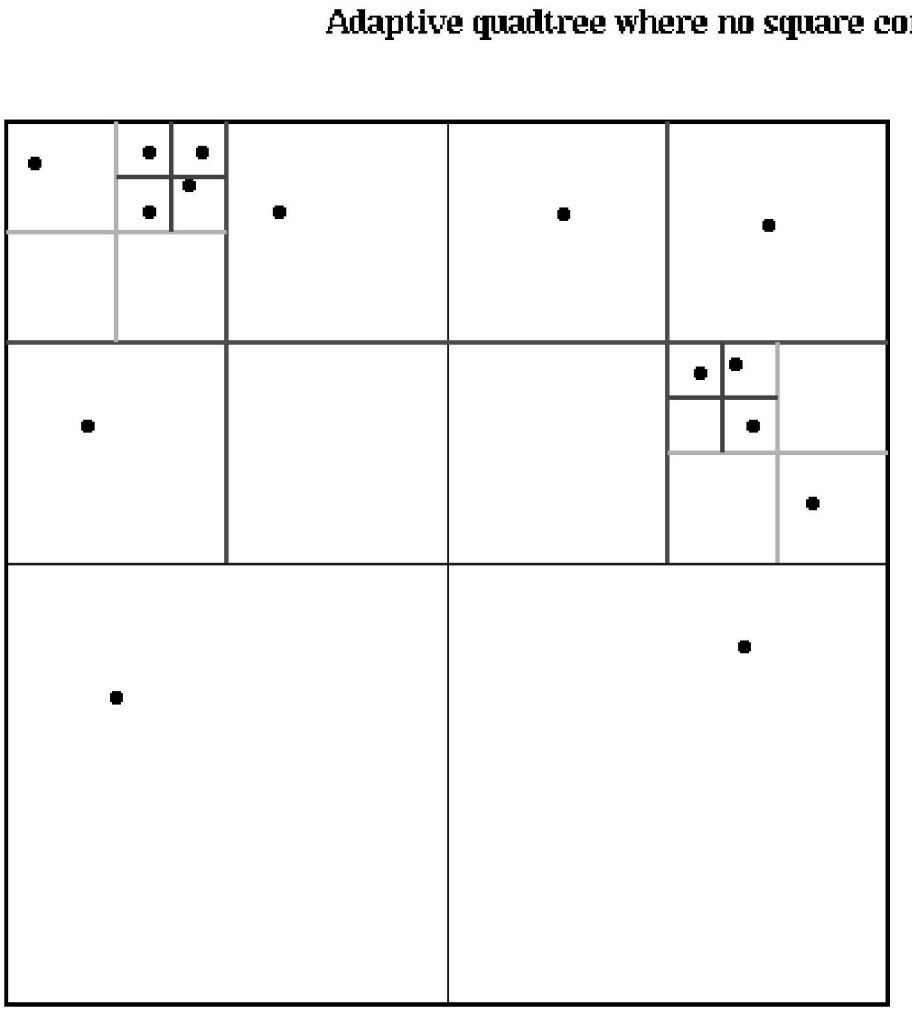
- **force calculation; contains a nested loop** — **Yes**
- **move operation; fixed amount of work per particle** — **No**
- **calculation of properties** — **Possible**

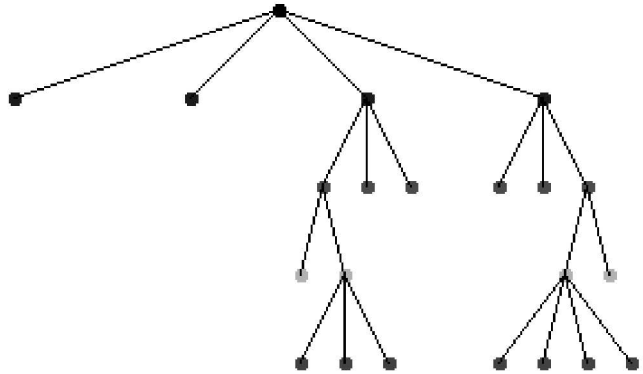
° **For high performance, focus on bottleneck force calculation**

° **Types of force to distinguish (in practice)** — **(in parallel code)**

- **External force** — **Trivial**
  - **Is independent on the other particles**
- **Short-range force** — **Easy**
  - **Depends on particles within fixed range**
- **Long-range force** — **Difficult**
  - **Depends on all other particles**

Example of an Adaptive Quad Tree





Child nodes enumerated counterclockwise from SW corner, empty ones are excluded

Adaptive Quad Tree Building Cost

- Cost ≤ N * maximum cost of a Quad_Tree_Insert
  = O(N * maximum depth of QuadTree)

- Uniform distribution:
  depth of QuadTree = O(log N),
  so Cost = O(N log N)

- Arbitrary distribution:
  depth of Quad Tree = O(b) = O(# bits in particle coordinates),
  so Cost = O(b N)
  - Note: depth of QuadTree b ≥ 1 + ^{4}log N

Barnes-Hut Algorithm

° **High Level description:**

1) Build the QuadTree using QuadTreeBuild
… already described, cost = O(N log N) or O(b N)

2) For each node, a sub-square in the QuadTree,
Compute the CM and total mass (TM) of all the particles it contains
… "post order traversal" of QuadTree, cost = O(N log N) or O(b N)

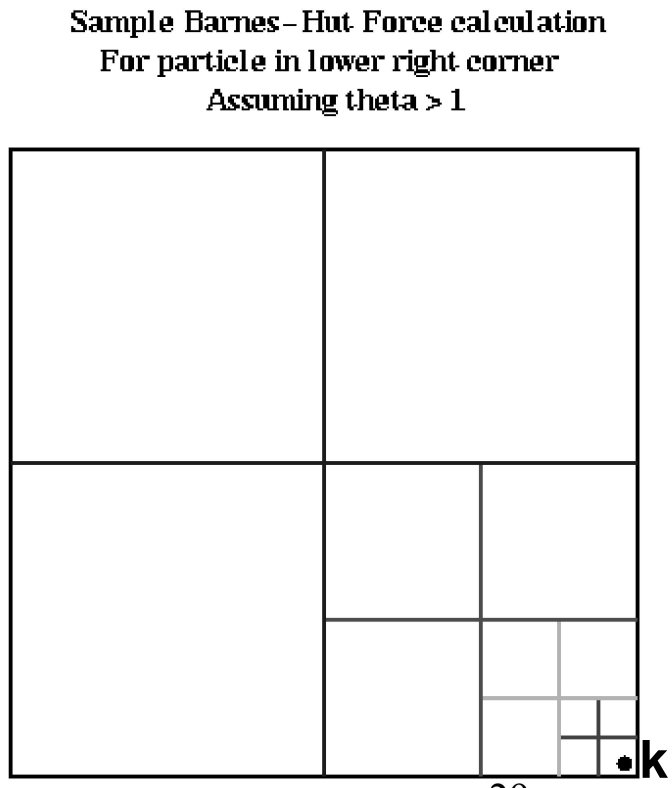
3) For each particle, traverse the QuadTree to compute the force on it,
using the CM and TM of "distant" sub-squares
… core of algorithm
… cost depends on accuracy desired but still O(N log N) or O(bN)

Total cost: **still O(N log N) or O(bN)**

→ Achieved order reduction per time step: $O(N^2)$ to O(N logN)

Step 3 of BH: Analysis

- **Correctness: recursive accumulation of force from each subtree**
  - **Each particle is accounted for exactly once, whether it is in a leaf or other node**
- **Complexity analysis**
  - **Cost of TreeForce(k, root) = O(depth in QuadTree of leaf containing k)**
  - **"Proof"; Example: Assume $\theta = 1$**
    - **For each undivided node, see fig., (except one containing k), $D/r < 1 < \theta$**
    - **There are only 3 nodes to consider at each level of the QuadTree, see fig.**
    - **There is O(1) work per node**
    - **Cost = O(level of k)**
  - **Total cost = $O(\Sigma_k$ level of k) = O(N log N)**
    - **Strongly depends on θ**



Fast Multiple Method (FMM)

- **"A fast algorithm for particle simulation", L. Greengard and V. Rokhlin, J. Comp. Phys. V. 73, 1987, many later papers**

- **Differences from Barnes-Hut**
  - **FMM computes the *potential* at every point, not just the force**
  - **FMM uses more information in each box than the CM and TM, so it is both more accurate and more expensive**
  - **In compensation, FMM accesses a fixed set of boxes at every level, independent of D/r**
  - **BH uses fixed information (CM and TM) in every box, but # boxes increases with accuracy.**
  - **FMM uses a fixed # boxes, but the amount of information per box increase with accuracy.**

Parallelizing Hierarchical N-Body codes

- **Barnes-Hut, FMM and related algorithms have similar computational structure:**
  - **1) Build the QuadTree**
  - **2) Traverse QuadTree from leaves to root and build outer expansions (just (TM,CM) for Barnes-Hut)**
  - **3) Traverse QuadTree from root to leaves and build any inner expansions**
  - **4) Traverse QuadTree to accumulate forces for each particle**

- **QuadTree changes dynamically when the particles move, so the tree has to be rebuilt (or adjusted) every time step.**
- **But: No doubly nested loop over all particles anywhere in the algorithm**

- **All 4 phases have to be parallelized efficiently.**

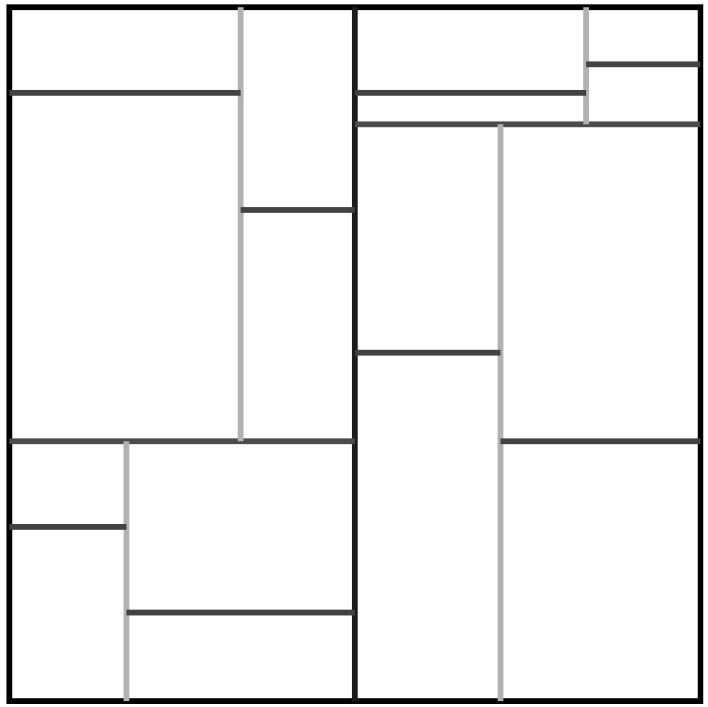
Parallelizing Hierachical N-Body codes

- **General idea: Domain decomposition**
  - **Assign regions of space to each processor**
  - **Regions may have different size or shape, to get a good load balance**
    - **Each region will have about N/p particles**
  - **Each processor will store part of QuadTree containing all particles (=leaves) in its region, and their ancestors in QuadTree**
    - **Root of tree and some generations stored by all processors, nodes may also be shared**
  - **Each processor will also store adjoining parts of QuadTree needed to compute forces for particles it owns**
    - **Subset of QuadTree needed by a processor called the Locally Essential Tree (LET)**
  - **Given the LET, all force accumulations (step 4)) are done in parallel, without communication**
- **Coarse grained parallelism**
  - **Each domain solves its own N-body problem, but somehow has to take into account the effect of all the other particles as well; like the ghost-points in the Poisson problem. Here those particles generate some "background" potential (in FMM)**

Load Balancing Scheme 1

- **Orthogonal Recursive Bisection (ORB) of space**
  - **Warren and Salmon, Supercomputing 92**
- **Recursively split region along axes into regions containing equal numbers of particles**
  - **Particles are grouped in rectangular regions; may be very elongated**
  - **No relation with tree**

Partitioning for 16 procs:



Load Balancing Scheme 2

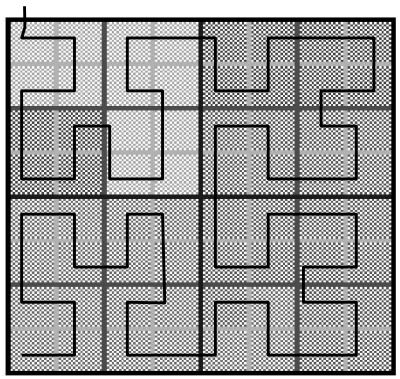
° **Idea: Partition QuadTree instead of space**
  - **Estimate work for each node, call total work W**
  - **Arrange nodes of QuadTree in some linear order (lots of choices)**
  - **Assign contiguous groups of nodes with work W/p to processors**

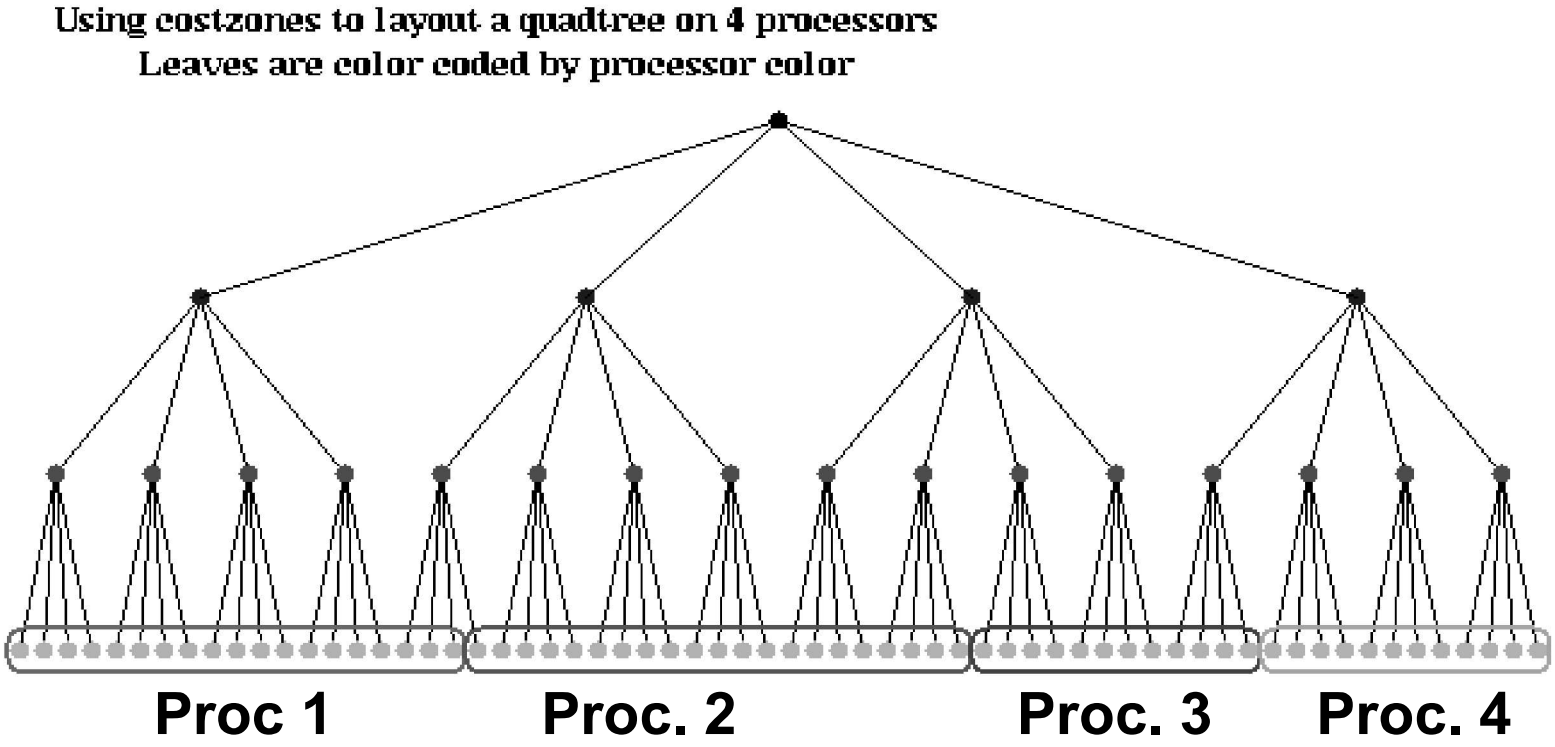
° **Method called: Costzones or Hashed Tree**
  - **J.P. Singh, PhD thesis, Stanford, 1993**
  - **Warren and Salmon, Supercomputing 93**

Load Balancing Scheme 2

° **Make sure that neighboring leaves in the tree are also neighboring in space**

- **Which of the 4 children of a node is "first" depends on the position of the node**
- **Orientation changes: clockwise / counter-clockwise**





Implementing Costzones

° **Random Sampling**

- **All processors own some particles,**
- **All processors send a small random sample of their particles to Processor 1**
- **Processor 1**
  - **builds small Quadtree serially,**
  - **determines its Costzones, and**
  - **broadcasts them to all processors**
- **Other processors build the part of Quadtree they are assigned by these Costzones**

° **All processors know all Costzones**

° **This is needed later to compute LET's**

Locally Essential Trees (LETs)

- **Warren and Salmon, 1992; Liu and Bhatt, 1994**
- **Definition:**
  - **A LET of a process is that part of the Tree that is necessary to compute the force on the particles that are owned by that process.**
- **Information about nodes near the root of the tree is present in all processes**
- **Information about nodes near some leaves of the tree that are all owned by a single process is needed in one or a few processes**

Computing Locally Essential Trees (LETs)

- Warren and Salmon, 1992; Liu and Bhatt, 1994
- Every processor needs a subset of the whole QuadTree, called the LET, to compute the force on all particles it owns
- Shared Memory
  - Receiver driven protocol
  - Each processor reads part of QuadTree it needs from shared memory on demand, keeps it in cache
  - Drawback: cache memory appears to need to grow proportionally to P to remain scalable
- Distributed Memory
  - Sender driven protocol
  - Each processor decides which other processors need parts of its local subset of the Quadtree, and sends these subsets

Locally Essential Trees in Distributed Memory

° **Barnes-Hut**

Which nodes are needed?

- **Let j and k be processes, n a node on process j**
- **Let D(n) be the side length of n**
- **Let $r_k(n)$ be the shortest distance from n to any point owned by k**
- **If either**

  (1) $D(n)/r_k(n) < \theta$ and $D(parent(n))/r_k(parent(n)) \geq \theta$, or

  (2) $D(n)/r_k(n) \geq \theta$

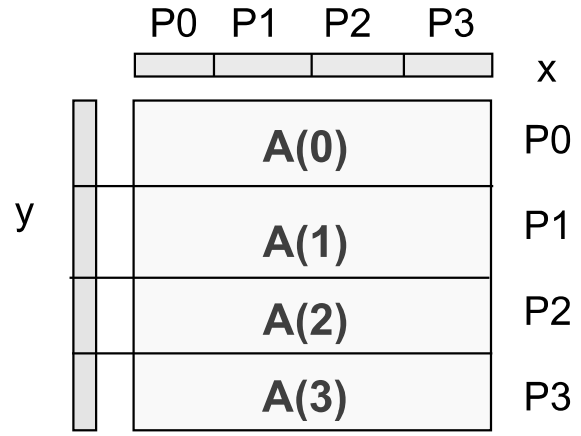
  then node n is part of k's LET, and so process j should send n to k

- **Condition (1) means (TM,CM) of n can be used on process k, but this is not true of any ancestor**
- **Condition (2) means that we need the ancestors of type (1) nodes too**

Parallel Matrix-Vector Product

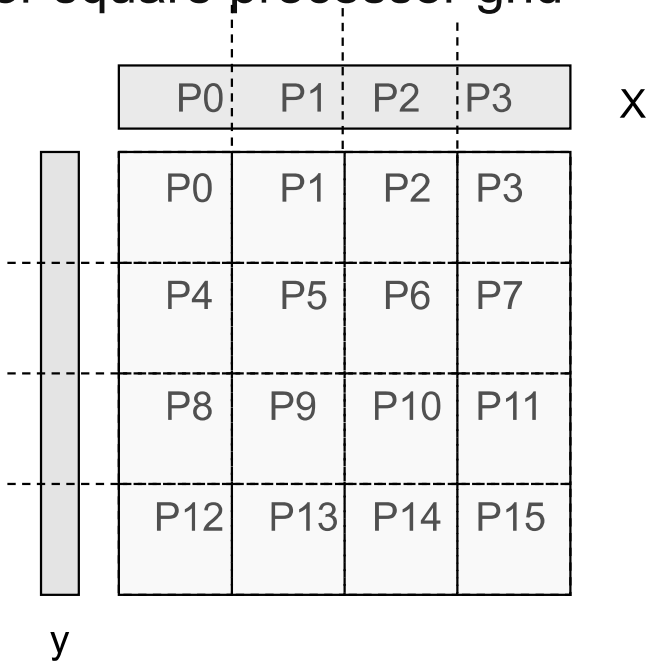
- Compute y = y + A*x, where A is a dense matrix
- Layout:
  - **1D row blocked**
- A(i) refers to the n by n/p block row that processor i owns,
- x(i) and y(i) similarly refer to segments of x,y owned by i
- **Algorithm:**
  - **Foreach processor i**
  - **Broadcast x(i)**
  - **Compute y(i) = A(i)*x**
- Algorithm uses the formula

  $y(i) = y(i) + A(i)*x = y(i) + \Sigma_j A(i,j)*x(j)$



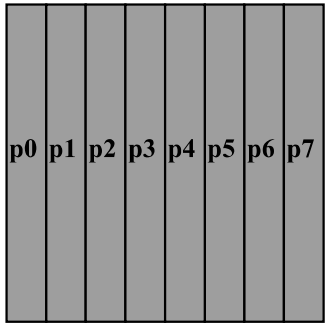
Matrix-Vector Product y = y + A*x

- A column layout of the matrix eliminates the broadcast of x
  - But adds a reduction to update the destination y
- A 2D blocked layout uses a broadcast and reduction, both on a subset of processors
  - sqrt(p) for square processor grid



Matrix Multiply with 1D Column Layout

- Assume matrices are n x n and n is divisible by p



May be a reasonable assumption for analysis, not for code

- A(i) refers to the n by n/p block column that processor i owns (similiarly for B(i) and C(i))
- B(i,j) is the n/p by n/p sublock of B(i)
  - in rows j*n/p through (j+1)*n/p - 1
- Algorithm uses the formula

  $C(i) = C(i) + A*B(i) = C(i) + \Sigma_j A(j)*B(j,i)$

MatMul: 1D layout on Bus without Broadcast

Naïve algorithm:

```
C(myproc) = C(myproc) + A(myproc)*B(myproc,myproc)
for i = 0 to p-1
  for j = 0 to p-1 except i
    if (myproc == i) send A(i) to processor j
    if (myproc == j)
        receive A(i) from processor i
        C(myproc) = C(myproc) + A(i)*B(i,myproc)
    barrier
```

Cost of inner loop:

computation: $2*n*(n/p)^2 = 2*n^3/p^2$

communication: $\alpha + \beta*n^2/p$

Naïve MatMul (continued)

Cost of inner loop:

computation: $2*n*(n/p)^2 = 2*n^3/p^2$

communication: $\alpha + \beta*n^2/p$

Data locality ratio ≈2*n/(β*p),
OK as long as n>>p, BUT

Only 1 pair of processors (i and j) are active on any iteration, and of those, only i is doing computation

=> the algorithm is almost entirely serial

Running time:

= (p*(p-1) + 1)*computation + p*(p-1)*communication

≈ $2*n^3 + p^2*\alpha + p*n^2*\beta$

This is worse than the serial time and grows with p.

Matmul for 1D layout on a Processor Ring

- Pairs of adjacent processors can communicate simultaneously

```
Copy A(myproc) into Tmp
C(myproc) = C(myproc) + Tmp*B(myproc , myproc)
for j = 1 to p-1
    Send Tmp to processor myproc+1 mod p
    Receive Tmp from processor myproc-1 mod p
    C(myproc) = C(myproc) + Tmp*B( myproc-j mod p , myproc)
```

- Same idea as for gravity in simple sharks and fish algorithm
  - May want double buffering in practice for overlap
  - Ignoring deadlock details in code
- Time of inner loop = $2*(\alpha + \beta*n^2/p) + 2*n*(n/p)^2$

Matmul for 1D layout on a Processor Ring

- Time of inner loop = $2*(\alpha + \beta*n^2/p) + 2*n*(n/p)^2$
- Total Time = $2*n*(n/p)^2$ + (p-1) * Time of inner loop
- $\approx 2*n^3/p + 2*p*\alpha + 2*\beta*n^2$
- (Nearly) Optimal for 1D layout on Ring or Bus, even with Broadcast:
  - Perfect speedup for arithmetic
  - A(myproc) must move to each other processor, costs at least
    (p-1)*cost of sending n*(n/p) words

- Parallel Efficiency = $2*n^3 / (p * \text{Total Time})$
  $= 1/(1 + \alpha * p^2/(2*n^3) + \beta * p/(2*n))$
  $= 1/ (1 + O(p/n))$
- Grows to 1 as n/p increases (or α and β shrink)

- But far from communication lower bound

SUMMA uses Outer Product form of MatMul

- C = A*B means $C(i,j) = \Sigma_k A(i,k)*B(k,j)$

- Column-wise outer product:

  C = A*B

  $= \Sigma_k A(:,k)*B(k,:)$

  $= \Sigma_k$ (k-th col of A)*(k-th row of B)

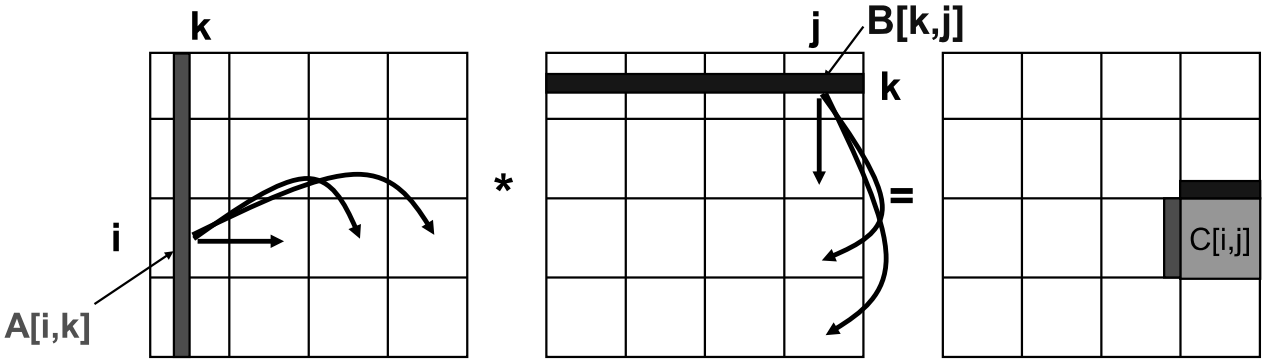
- Block column-wise outer product
  (block size = 4 for illustration)

  C = A*B

  = A(:,1:4)*B(1:4,:) + A(:,5:8)*B(5:8,:) + …

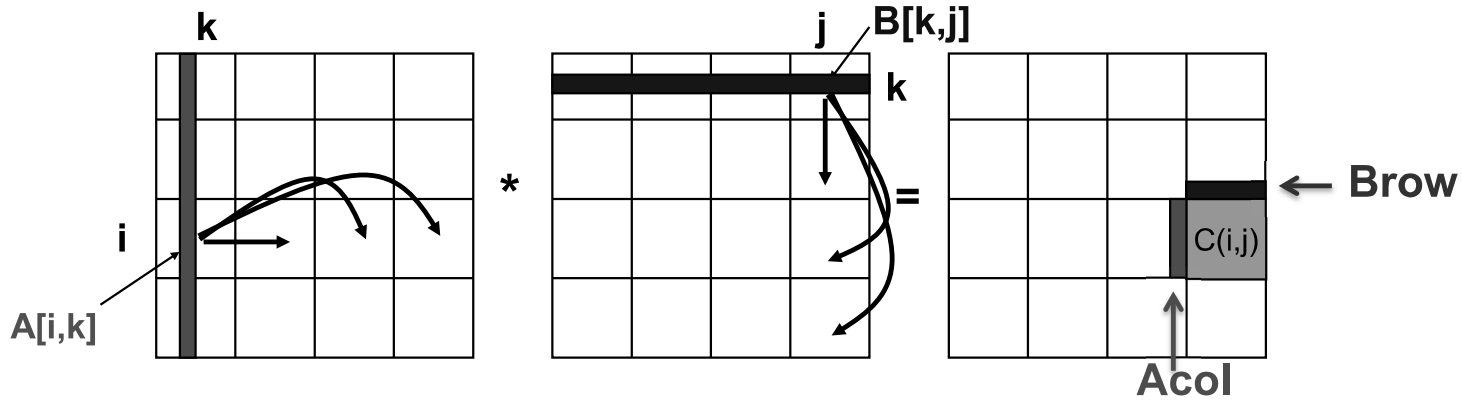
  $= \Sigma_k$ (k-th block of 4 cols of A)*
  (k-th block of 4 rows of B)

SUMMA – n x n matmul on $P^{1/2}$ x $P^{1/2}$ grid



- C[i, j] is $n/P^{1/2}$ x $n/P^{1/2}$ submatrix of C on processor P_{ij}
- A[i,k] is $n/P^{1/2}$ x b submatrix of A
- B[k,j] is b x $n/P^{1/2}$ submatrix of B
- C[i,j] = C[i,j] + $\Sigma_{\mathbf{k}}$ A[i,k]*B[k,j]
  - summation over submatrices
- Need not be square processor grid

SUMMA– n x n matmul on $P^{1/2}$ x $P^{1/2}$ grid



For k=0 to n/b-1

for all i = 1 to $P^{1/2}$

owner of A[i,k] broadcasts it to whole processor row (using binary tree)

for all j = 1 to $P^{1/2}$

owner of B[k,j] broadcasts it to whole processor column (using bin. tree)

Receive A[i,k] into Acol

Receive B[k,j] into Brow

C_myproc = C_myproc + Acol * Brow

SUMMA Costs

```
For k=0 to n/b-1
    for all i = 1 to P^1/2
          owner of A[i,k] broadcasts it to whole processor row (using binary tree)
           … #words  = log P^1/2 *b*n/P^1/2 ,     #messages = log P^1/2
    for all j = 1 to  P^1/2
          owner of B[k,j] broadcasts it to whole processor column (using bin. tree)
           …  same #words and #messages
    Receive A[i,k] into Acol
    Receive B[k,j] into Brow
    C_myproc = C_myproc + Acol * Brow     … #flops = 2n^2*b/P
```

- Total #words = $\log P * n^2 / P^{1/2}$ versus 2*n/P on page 17
  - Within factor of log P of lower bound
  - (more complicated implementation removes log P factor)
- Total #messages = $\log P * n/b$
  - Choose b close to maximum, $n/P^{1/2}$, to approach lower bound $P^{1/2}$
- Total #flops = $2n^3/P$

Some communication lower bounds of matrix multiply

- Lower bound assumed 1 copy of data: $M = O(n^2/P)$ per proc.
- What if matrix small enough to fit c>1 copies, so $M = cn^2/P$?
  - #words_moved = $\Omega(\ \#flops / M^{1/2}\ ) = \Omega(\ n^2 / (\ c^{1/2} P^{1/2}\ ))$
  - #messages = $\Omega(\ \#flops / M^{3/2}\ ) = \Omega(\ P^{1/2} / c^{3/2})$

- Can we attain new lower bound?
  - Special case: "3D Matmul": $c = P^{1/3}$
    - Bernsten 89, Agarwal, Chandra, Snir 90, Aggarwal 95
    - Processors arranged in $P^{1/3} \times P^{1/3} \times P^{1/3}$ grid
    - Processor (i,j,k) performs C(i,j) = C(i,j) + A(i,k)*B(k,j), where each submatrix is $n/P^{1/3} \times n/P^{1/3}$
  - Not always that much memory available…

No further discussions

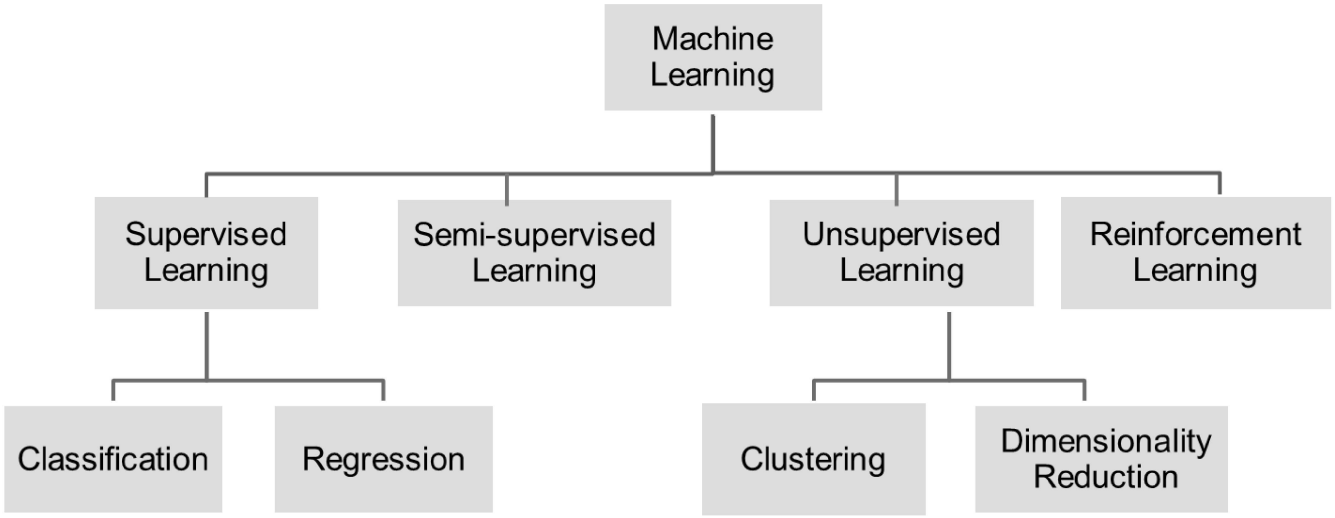
I. Learning from Data

Supervised learning: **involves building a statistical model for predicting, or estimating, an *output* based on one or more *inputs*. (e.g. linear regression alg. and SVM)**

Unsupervised learning: **there are inputs but no supervising output; nevertheless we can learn relationships and structure from such data. (e.g., *K*-means clustering alg.)**

Regression: **predicting a *continuous* or *quantitative* output value.**

Classification: **predict a non-numerical value—that is, a *categorical* or *qualitative* output.**



Parallelism in Machine Learning

Implicit Parallelization: Keep the overall algorithm structure (the sequence of operations) intact and parallelize the individual operations.

Example: parallelizing the BLAS operations in previous figure

+ Often achieves exactly the same accuracy (e.g., model parallelism in DNN training)

- Scalability can be limited if the critical path of the algorithm is long

Explicit Parallelization: Modify the algorithm to extract more parallelism, such as working on individual pieces whose results can later be combined

Examples: CA-SVM and data parallelism in DNNs

+ Significantly better scalability can be achieved

- No longer the same algorithmic properties (e.g. HogWild!).

Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent, F. Niu et al, NIPS 2011

Slides source: Aidyn Buluc

Parallelization Opportunities

Different from the data parallelism with 'owner-compute' rule (distribute output)

1. **Data parallelism**

   Distribute the input (sets of images, text, audio, etc.)

   a) **Batch parallelism**

   - Distribute a group
     processor
   - *When people mer*
     this is what they n

   b) **Domain parallelism**

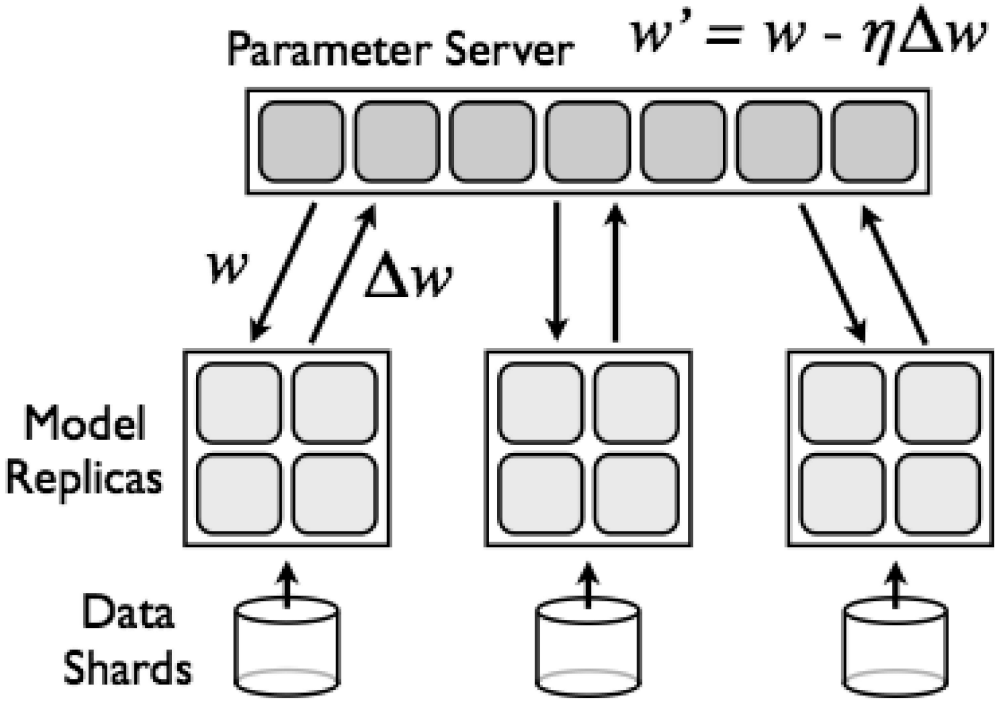
   - Data points/features of individual sample are subdivided and distribute parts to processors.

2. **Model parallelism:**

   Distribute the neural network (i.e. its weights)

Epoch: the entire training set is used once;
Mini-batch: a subset of training samples, weights are updated once the entire mini-batch is used.
SDG: update the weights for every training sample (mini-batch size=1)

Batch Parallelism #1



Parameter server is some sort of master process

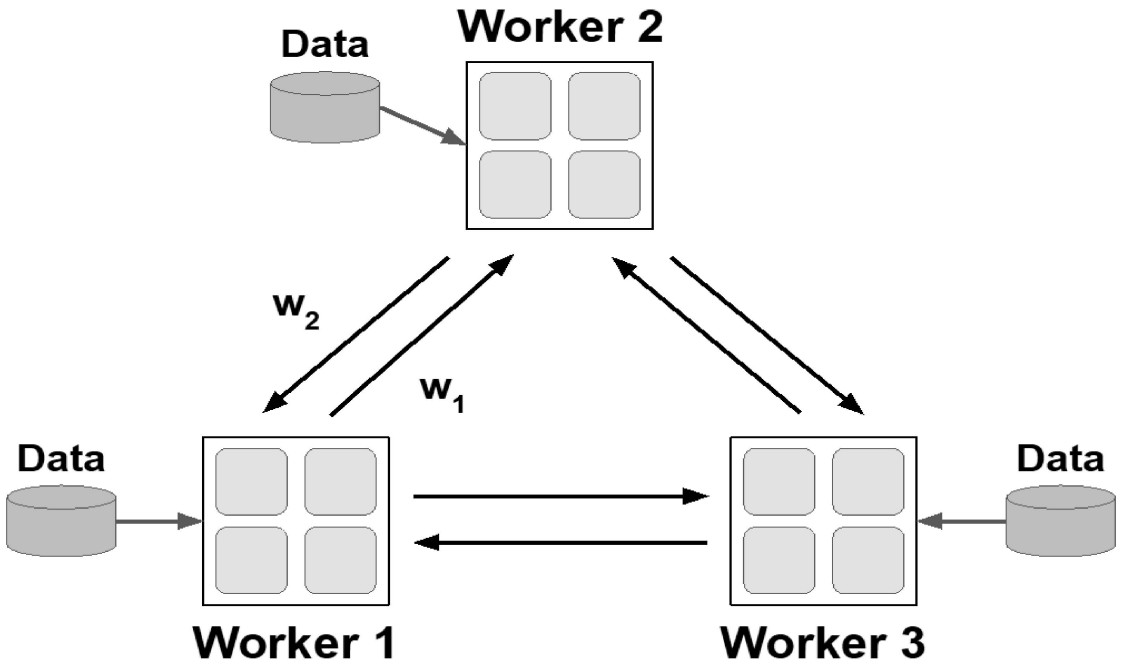
- The fetching and updating of gradients in the parameter server can be done either ***synchronously*** or ***asynchronously***.
- Both has pros and cons. Over-synchronization hurts performance where asynchrony is not-reproducible and might hurt convergence

Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems. 2012.

Batch Parallelism #2

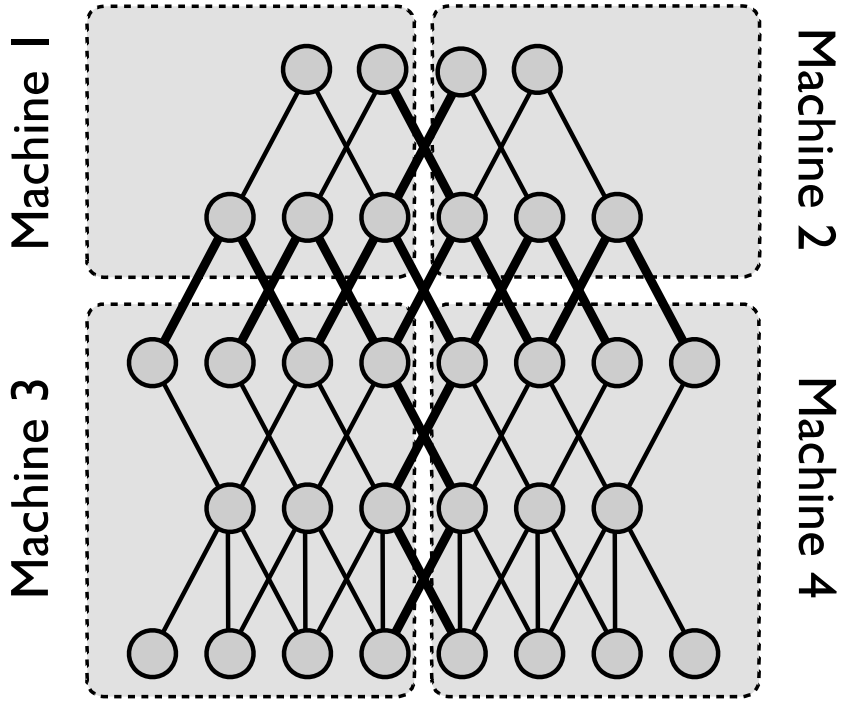
Options to avoid the parameter server bottleneck

1. **For synchronous SGD:** Perform all-reduce over the network to update gradients (good old MPI_Allreduce)
2. **For asynchronous SGD:** Peer-to-peer gossiping



Peter Jin, Forrest Iandola, Kurt Keutzer, “How to scale distributed deep learning?” NIPS ML Sys 2016

Model Parallelism



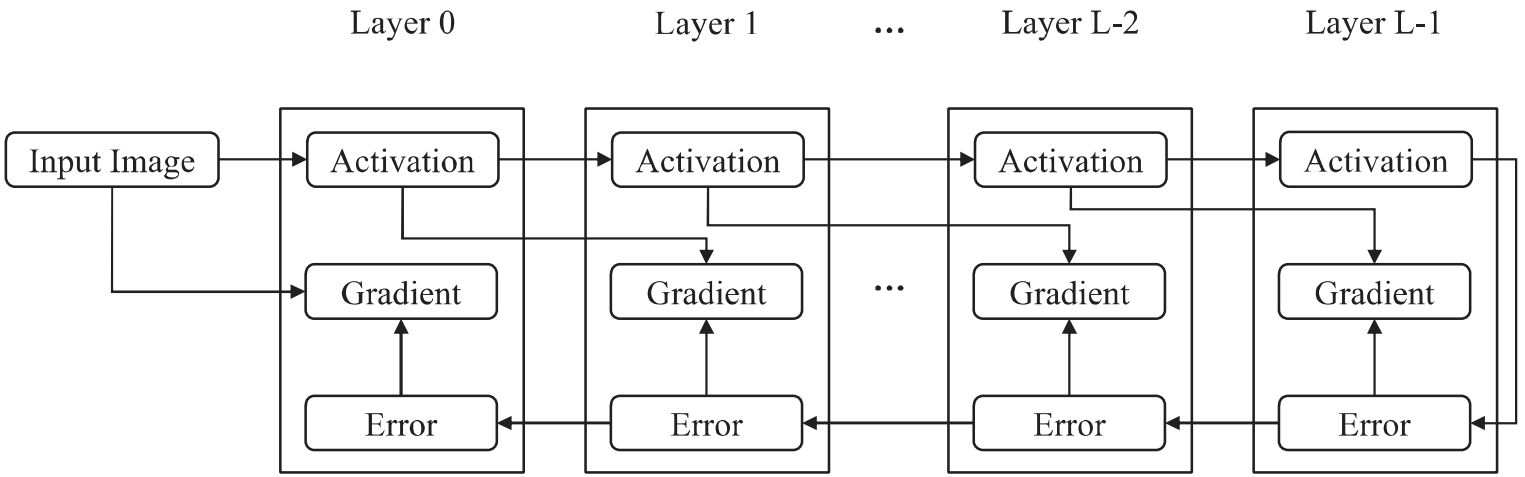
(inter layer parallelism is better called pipelining)

Interpretation #1: Partition your neural network into processors
Interpretation #2: Perform your matrix operations in parallel

Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems. 2012.

The overlapping opportunities

- In backpropagation, the errors are propagated from the last layer to the first layer and data dependency exists between any two consecutive layers.
- In contrast, the gradients in different layers are independent of each other.



Activations are propagated from left to right in forward stage; errors are propagated from right to left in backpropagation stage. Gradients are computed using the activations and errors. Arrows indicated data dependencies.

S. Lee, et al. "Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication.", HiPC 2017

Algorithm 1 Mini-Batch SGD CNN Training Algorithm
(M: the number of mini-batches, L: the number of layers

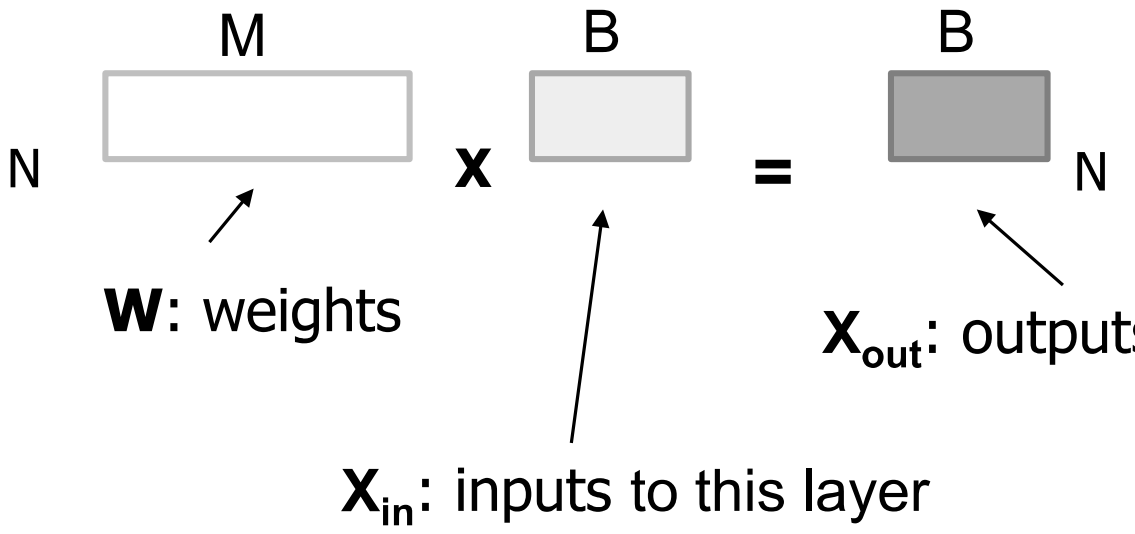
1: **for** each mini batch $m = 0, ...M-1$ **do**
2: Initialize $\Delta W = 0$
3: Get the m^{th} mini batch, D^m.
4: **for** each layer $l = 0, ...L-1$ **do**
5: Calculate activations A^l based on D^m.
6: **for** each layer $l = L-1, ...0$ **do**
7: Calculate errors E^l.
8: Calculate weight gradients ΔW^l.
9: **for** each layer $l = 0, ...L-1$ **do**
10: Update parameters, W^l and B^l.

Forward: $a_n^l = \sigma(\sum_{i=0}^{|W|-1} w_i^l a_{n+i}^{l-1} + b_n^l)$

Backward: $e_n^l = \sum_{i=0}^{|W|-1} w_i^{l+1} e_{n-i}^{l+1},$

where a_n^l, b_n^l, and e_n^l are the n^{th} activation, bias, and error in layer l respectively, w_i^l is an weight on the i^{th} connection between layer l and layer $l-1$, $|W|$ is the number of weights in a feature map, and σ is the activation function.

SGD training of NNs as matrix operations

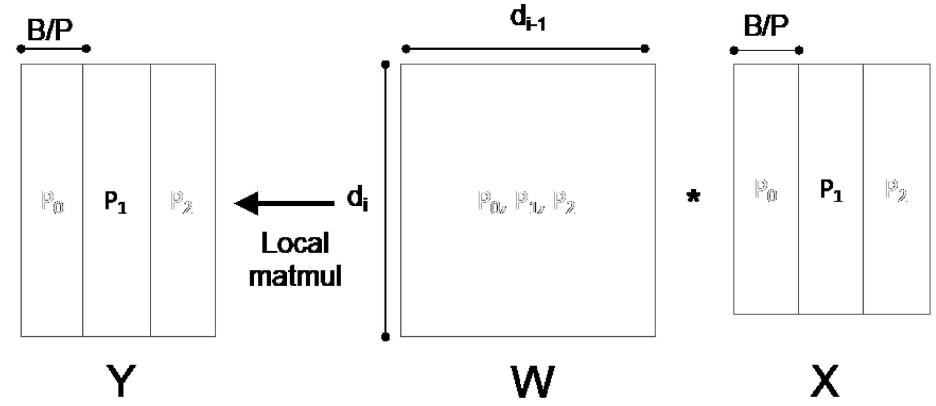


N = the number of outputs
M = the number of inputs
B = the size of the minibatch

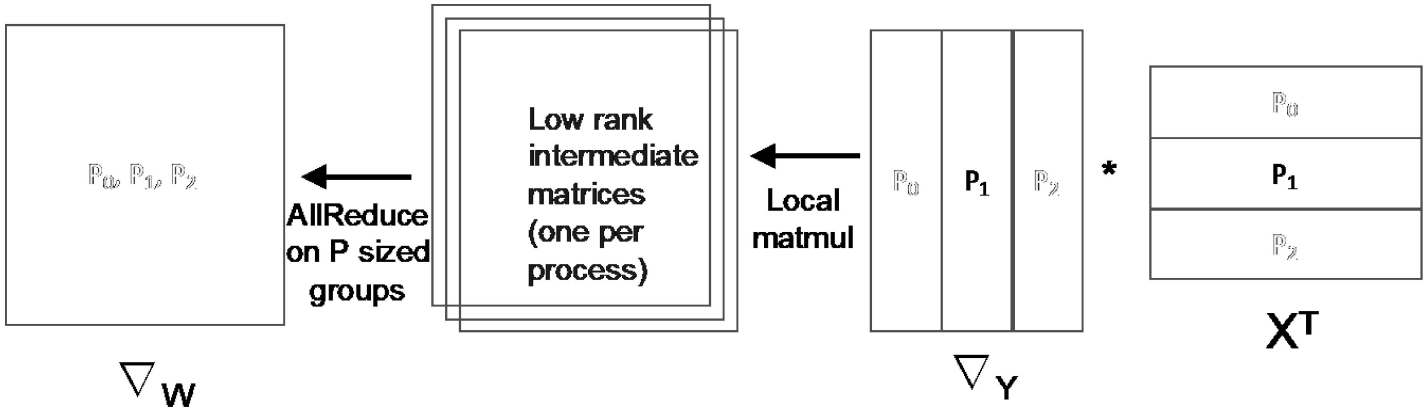
The impact to parallelism:

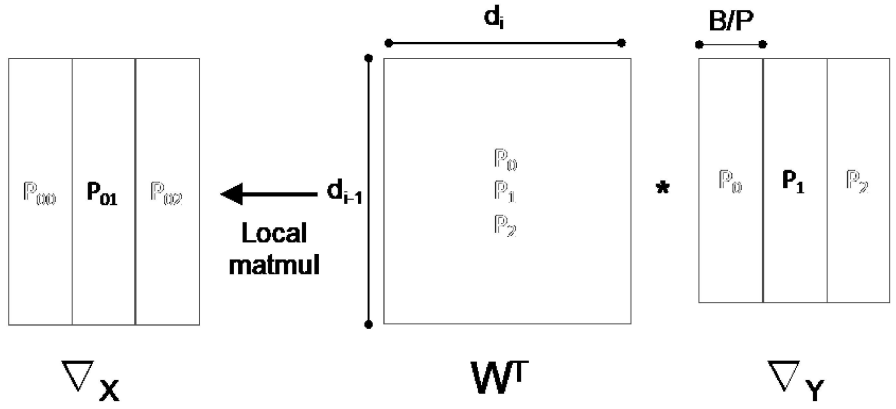
- **W** is replicated to processor, so it doesn't change
- $\mathbf{X_{in}}$ and $\mathbf{X_{out}}$ gets skinnier if we only use data parallelism, i.e. distributing **b=B/p** mini-batches per processor
- GEMM (i.e. matrix-matrix multiply) performance suffers as *matrix dimensions get smaller* and *more skewed*
- **Result:** Data parallelism can hurt single-node performance

Data Parallel SGD training of NNs as matrix operations



1. Which matrices are replicated?
2. Where is the communication?
3. Which steps can be overlapped?





$\nabla_Y = \partial L / \partial Y$ = how did the lost function change as output activations change?

$\nabla_X = \partial L / \partial X$

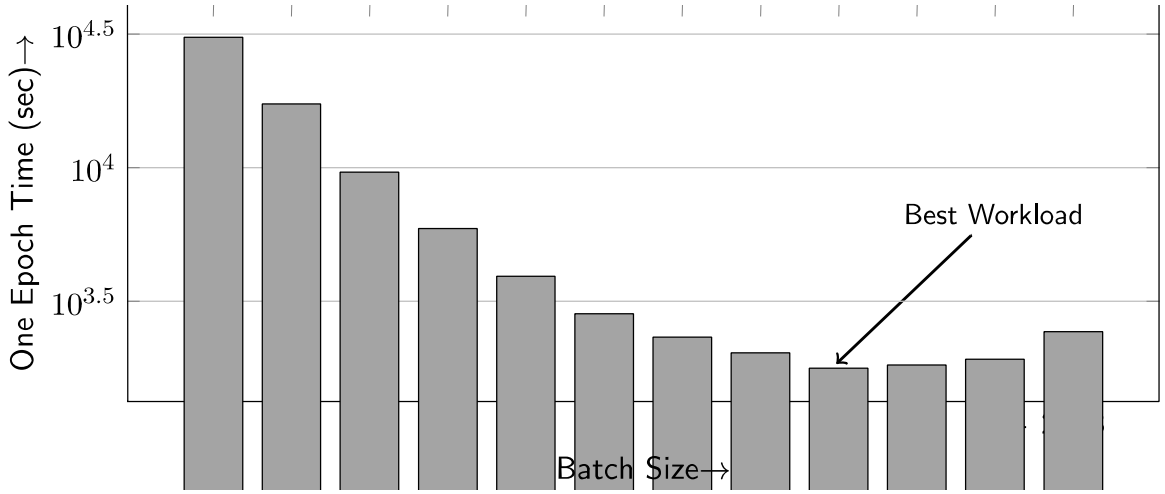
$\nabla_W = \partial L / \partial W$

Batch Parallel Strong Scaling

- **Per-iteration communication cost of batch parallelism is independent of the batch size:**
  - **larger batch → less communication per epoch (full pass over the data set)**

$$T_{comm}(batch) = 2\sum_{i=0}^{L}\left(\alpha\log(P) + \beta\frac{P-1}{P}|W_i|\right)$$

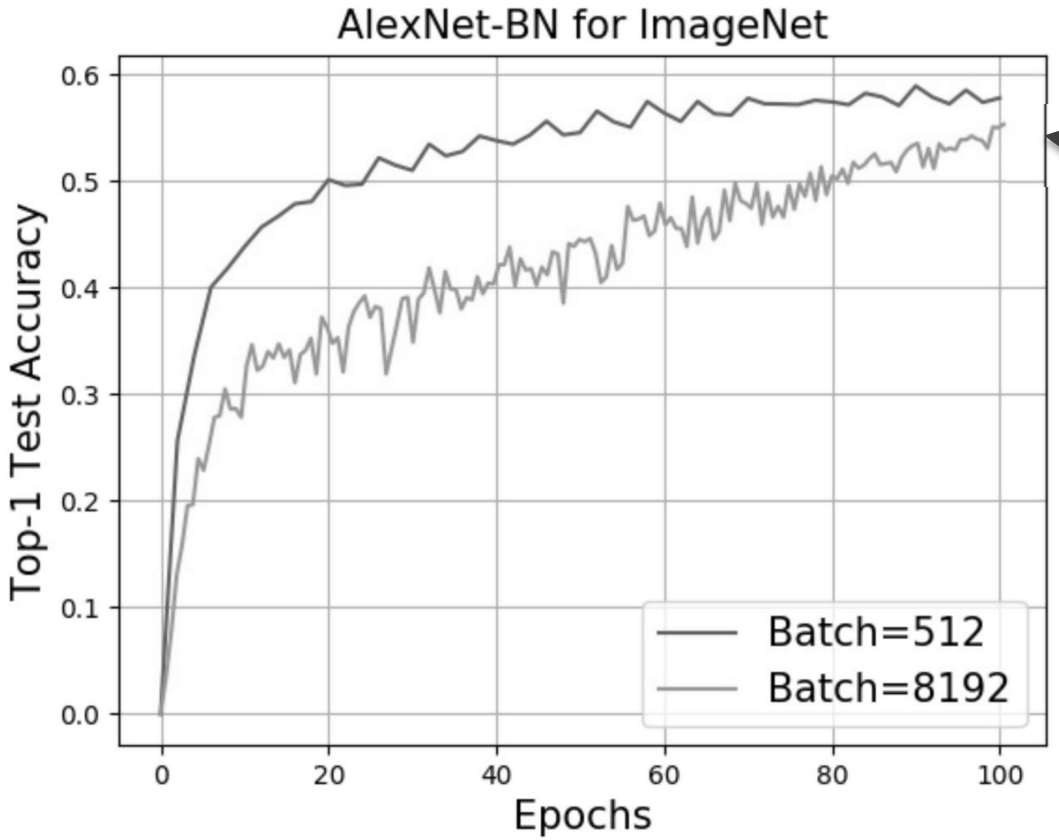
- But processor utilization goes down significantly for P>>1
  - Result: Batch parallel has poor strong scaling



One epoch training time of AlexNet computed on a single KNL

Problems with Batch Parallelism

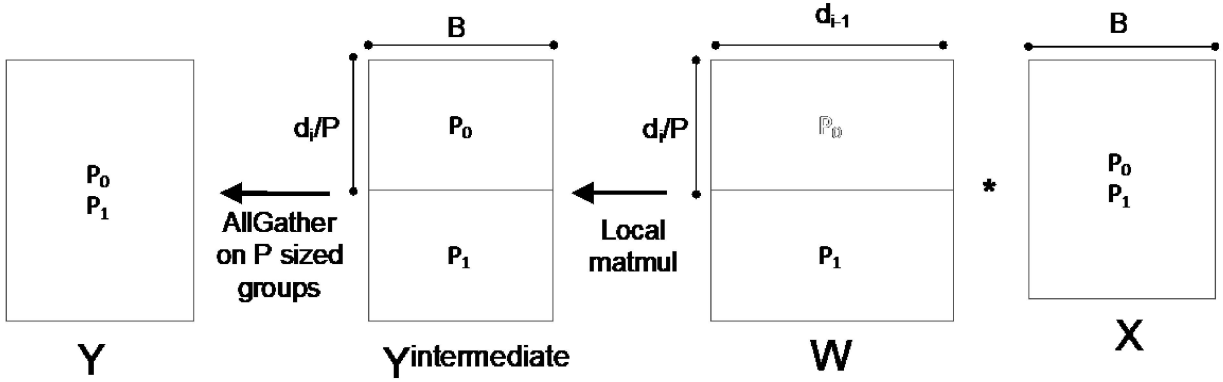
- Batch parallel scaling is limited to B
  - Larger Batch -> higher strong scaling efficiency
- But SGD does not perform well for large batch

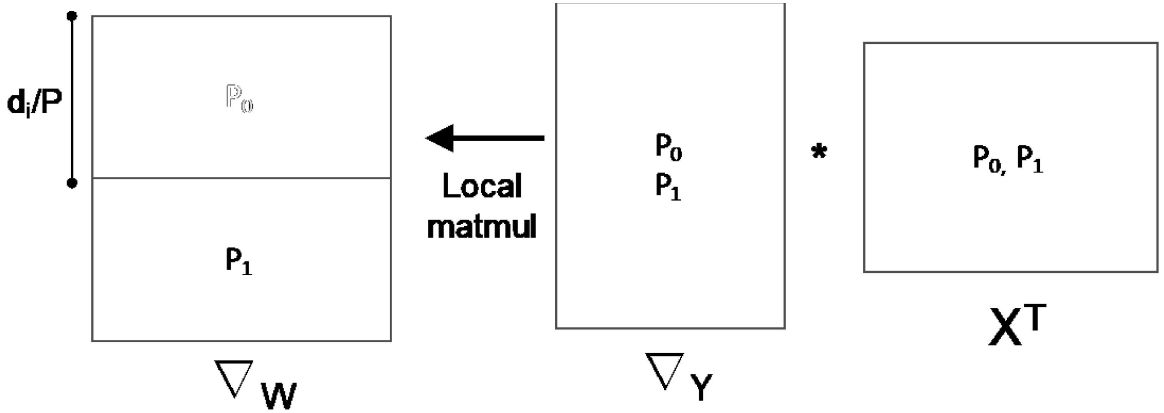


Performance Gap

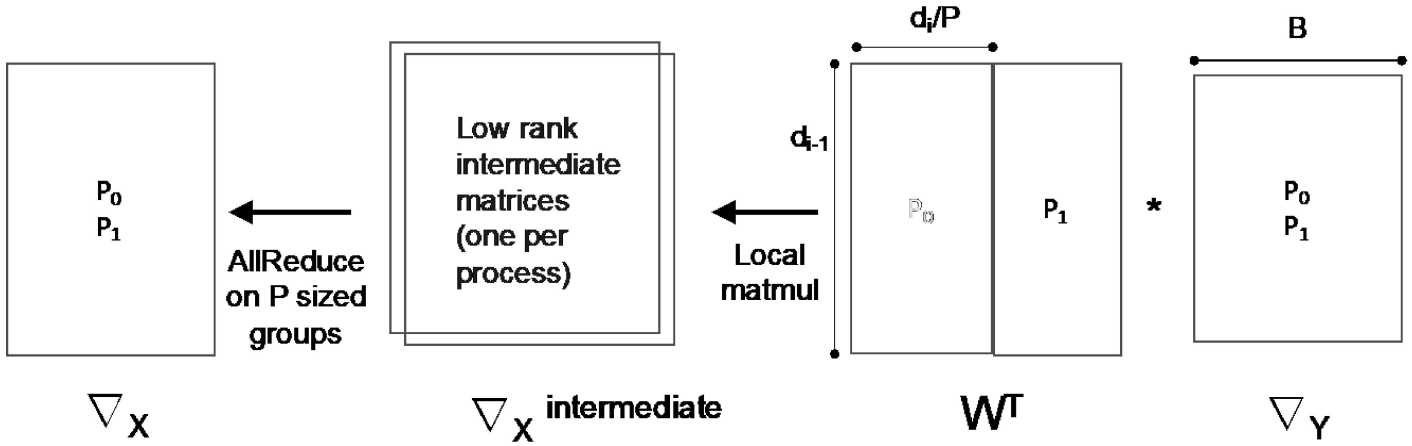
Ginsburg, Boris, Igor Gitman, and Yang You. "Large Batch Training of Convolutional Networks with Layer-wise Adaptive Rate Scaling." (2018).

Model Parallel SGD training of NNs as matrix operations





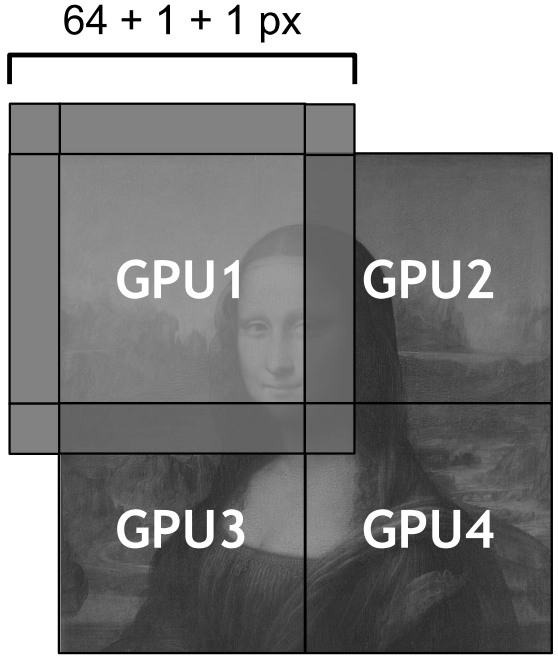
1. Which matrices are replicated?
2. Where is the communication?
3. How can matrix algebra capture both model and data parallelism?



Communication Complexity of Domain Parallel

- Additional communication for halo exchange during forward and backwards pass
  - Negligible cost for early layers for which activation size is large (i.e. convolutional)

$$T_{comm}(domain) = \sum_{i=0}^{L} \left(\alpha + \beta B X_W^i X_C^i k_h^i / 2\right)$$
$$+ \sum_{i=0}^{L} \left(\alpha + \beta B Y_W^i Y_C^i k_w^i / 2\right)$$
$$+ 2 \sum_{i=0}^{L} \left(\alpha \log(P) + \beta \frac{P-1}{P} |W_i|\right)$$



1. Gradient Descent Optimization

Batch Optimization: Update 'weights' w_t by mean gradient generated by ALL samples.

Algorithm 1 BATCH optimization with samples $X = \{x_0, \ldots, x_m\}$, iterations T and states w

1: **for all** $t = 0 \ldots T$ **do**
2: **Init** $w_{t+1} = 0$ w_t
3: **for all** $x_j \in X$ **do**
4: **aggregate** $w_{t+1} = w_{t+1} + \partial_w x_j(w_t)$
5: $w_{t+1} = w_{t+1}/|X|$

Partial derivative w.r.t. x_j: $\Delta_j(w_t) := \partial_w x_j(w_t)$

Stochastic Gradient Descent (SGD):

Update 'weights' w_t for every single sample. (stochastic: select a training sample at random)

Algorithm 2 SGD with samples $X = \{x_0, \ldots, x_m\}$, iterations T, steps size ϵ and states w

Require: $\epsilon > 0$

1: **for all** $t = 0 \ldots T$ **do**
2: **draw** $j \in \{1 \ldots m\}$ uniformly at random
3: **update** $w_{t+1} \leftarrow w_t - \epsilon \partial_w x_j(w_t)$
4: **return** w_{T+1}

Mini-batch SGD:

Update weights w_t after a group of training smaples (mini-batch).

Algorithm 4 Mini-Batch SGD with samples $X = \{x_0, \ldots, x_m\}$, iterations T, steps size ϵ, number of threads n and mini-batch size b

Require: $\epsilon > 0$

1: **for all** $t = 0 \ldots T$ **do**
2: **draw** mini-batch $M \leftarrow b$ samples from X
3: **Init** $\Delta w_t = 0$
4: **for all** $x \in M$ **do**
5: **aggregate update** $\Delta w \leftarrow \partial_w x_j(w_t)$
6: **update** $w_{t+1} \leftarrow w_t - \epsilon \Delta w_t$
7: **return** w_{T+1}

2. Parallel SGD *n* nodes/threads operate independently until convergence.

Algorithm 3 SimuParallelSGD with samples $X = \{x_0, \ldots, x_m\}$, iterations T, steps size ϵ, number of nodes n and states w

Require: $\epsilon > 0, n > 1$

define $H = \lfloor \frac{m}{n} \rfloor$
randomly **partition** X, giving H samples to each node
for all $i \in \{1, \ldots, n\}$ **parallel do**
    randomly **shuffle** samples on node i
    init $w_0^i = 0$
    for all $t = 0 \ldots T$ **do**
        get the tth sample on the ith node
        update $w_{t+1}^i \leftarrow w_t^i - \epsilon \Delta_t(w_t^i)$
aggregate $v = \frac{1}{n} \sum_{i=1}^{n} w_t^i$
return v

M. Zinkevich, et al, Parallelized stochastic gradient descent. In Proc. NIPS 2010, pp. 2595-2603.

Asynchronous SGD

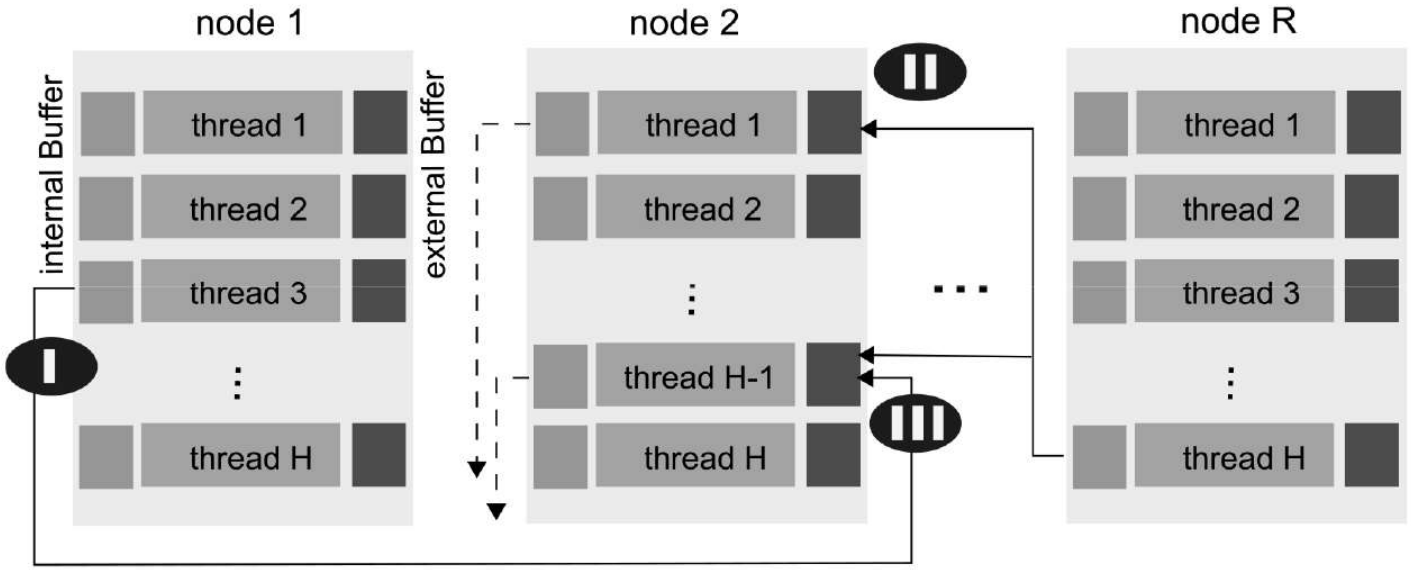


Figure 2: Overview of the asynchronous update communication used in ASGD. Given a cluster environment of R nodes with H threads each, the blue markers indicate different stages and scenarios of the communication mode. I: Thread 3 of node 1 finished the computation of of its local mini-batch update. The external buffer is empty. Hence it executes the update locally and sends the resulting state to a few random recipients. II: Thread 1 of node 2 receives an update. When its local mini-batch update is ready, it will use the external buffer to correct its local update and then follow I. III: Shows a potential data race: two external updates might overlap in the external buffer of thread $H-1$ of node 2. Resolving data races is discussed in section 4.4.

3. ASGD

Algorithm 5 ASGD $(X = \{x_0, \ldots, x_m\}, T, \epsilon, w_0, b)$

Require: $\epsilon > 0, n > 1$

define $H = \lfloor \frac{m}{n} \rfloor$
randomly **partition** X, giving H samples to each node
for all $i \in \{1, \ldots, n\}$ **parallel do**
    randomly **shuffle** samples on node i
    init $w_0^i = 0$
    for all $t = 0 \ldots T$ **do**
        draw mini-batch $M \leftarrow \overline{b \text{ samples from } X}$
        update $w_{t+1}^i \leftarrow w_t^i - \epsilon \overline{\boldsymbol{\Delta}_M(w_{t+1}^i)}$
        send w_{t+1}^i to random node $\neq i$
return w_I^1

ASGD updating: handling of multiple updates in one iteration at node *i*

To combine with the communicated state $w_{t'}^{j}$ from an unknown iteration *t'* of some random node j:

$$\overline{\Delta_t(w_{t+1}^i)} = \frac{1}{2}\left(w_t^i + w_{t'}^j\right) + \Delta_t(w_{t+1}^i) \qquad (2)$$

For the usage of N external buffers per thread, we generalize equation (2) to:

$$\overline{\Delta_t(w_{t+1}^i)} = w_t^i - \frac{1}{|N|+1}\left(\sum_{n=1}^{N}(w_{t'}^n) + w_t^i\right) + \Delta_t(w_{t+1}^i),$$

$$\text{where } |N| := \sum_{n=0}^{N}\lambda(w_{t'}^n), \quad \lambda(w_{t'}^n) = \begin{cases} 1 & \text{if } \|w_{t'}^n\|_2 > 0 \\ 0 & \text{otherwise} \end{cases}$$

Parzen-window: to include only the "good" update from other threads:

$$\delta(i,j) := \begin{cases} 1 & \text{if } \|(w_t^i - \epsilon\Delta w_t^i) - w_{t'}^j\|^2 < \|w_t^i - w_{t'}^j\|^2 \\ 0 & \text{otherwise} \end{cases}$$

ASGD updating: illustration

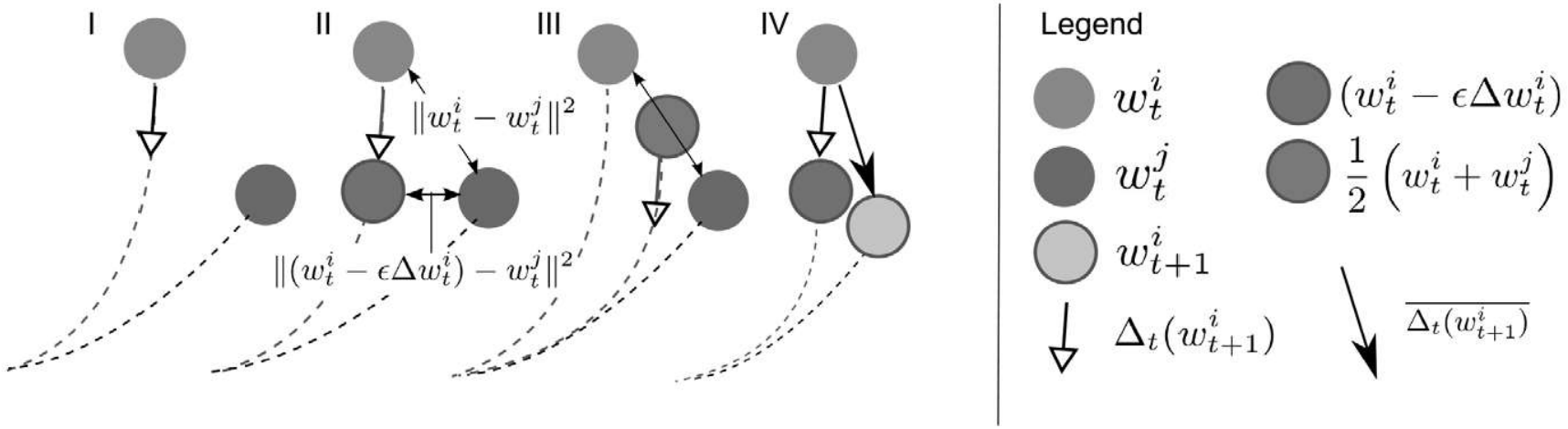


Figure 4: ASGD updating. This figure visualizes the update algorithm of a process with state w_t^i, its local mini-batch update $\Delta_t(w_{t+1}^i)$ and received external state w_t^j for a simplified 1-dimensional optimization problem. The dotted lines indicate a projection of the expected descent path to an (local) optimum. I: Initial setting: $\boldsymbol{\Delta}_M(w_{t+1}^i)$ is computed and w_t^j is in the external buffer. II: Parzen-window masking of w_t^j. Only if the condition of equation (4) is met, w_t^j will contribute to the local update. III: Computing $\overline{\boldsymbol{\Delta}_M(w_{t+1}^i)}$. IV: Updating $w_{t+1}^i \leftarrow w_t^i - \epsilon\overline{\boldsymbol{\Delta}_M(w_{t+1}^i)}$.

4. Performance

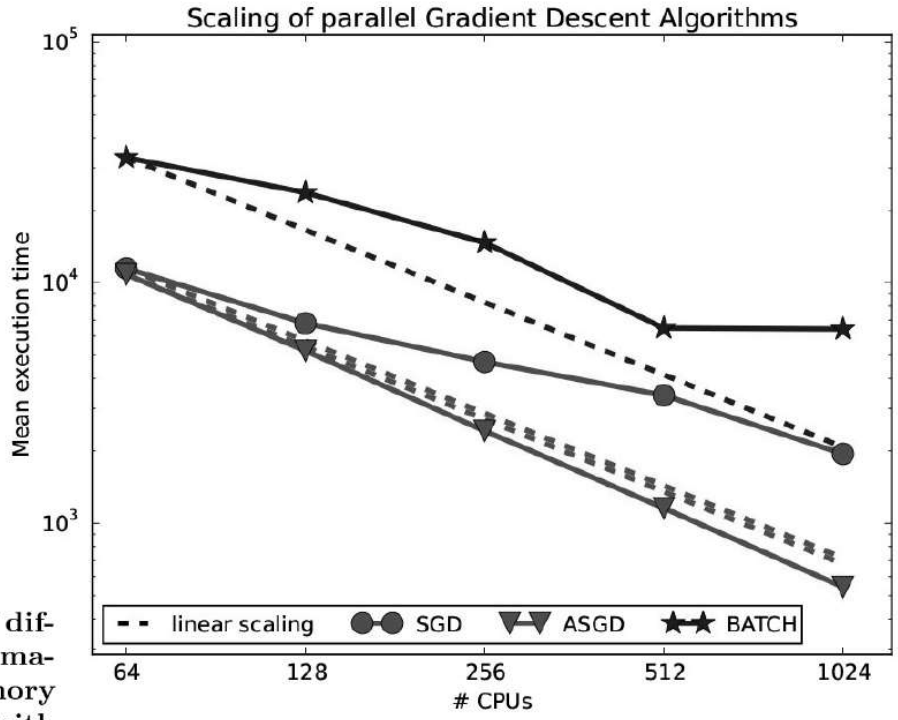


Figure 1: Evaluation of the scaling properties of different parallel gradient descent algorithms for machine learning applications on distributed memory sytems. Results show a K-Means clustering with k=10 on a 10-dimensional target space, represented by ~1TB of training samples. Our novel ASGD method is not only the fastest algorithm in this test, it also shows better than linear scaling performance. Outperforming the SGD parallelization by [19] and the MapReduce based BATCH [5] optimization, which both suffer from communication overheads.

Asynchronous communication

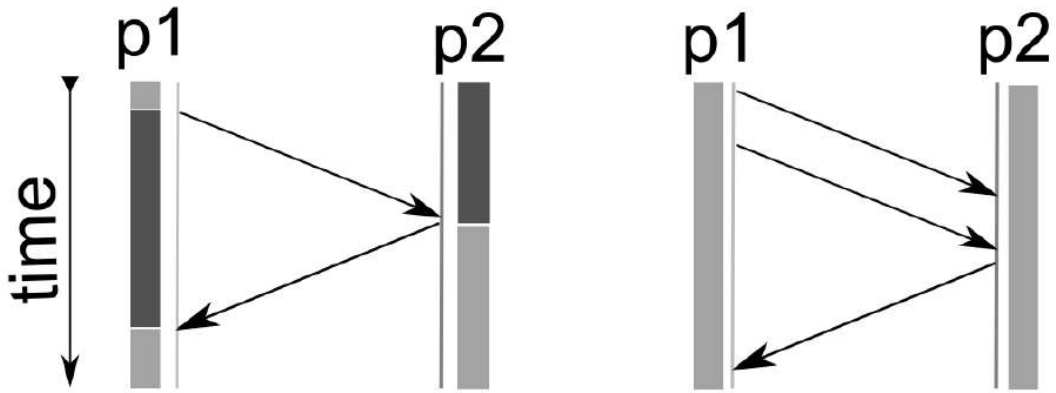


Figure 3: Single-sided asynchronous communication model (right) compared to a typical synchronous model (left). The red areas indicate dependency locks of the processes $p1, p2$, waiting for data or acknowledgements. The asynchronous model is lock-free, but comes at the price that processes never know if and when and in what order messages reach the receiver. Hence, a process can only be informed about past states of a remote computation, never about the current status.

Remark:
Implementation: one-sided put and get, or asyn. communication.

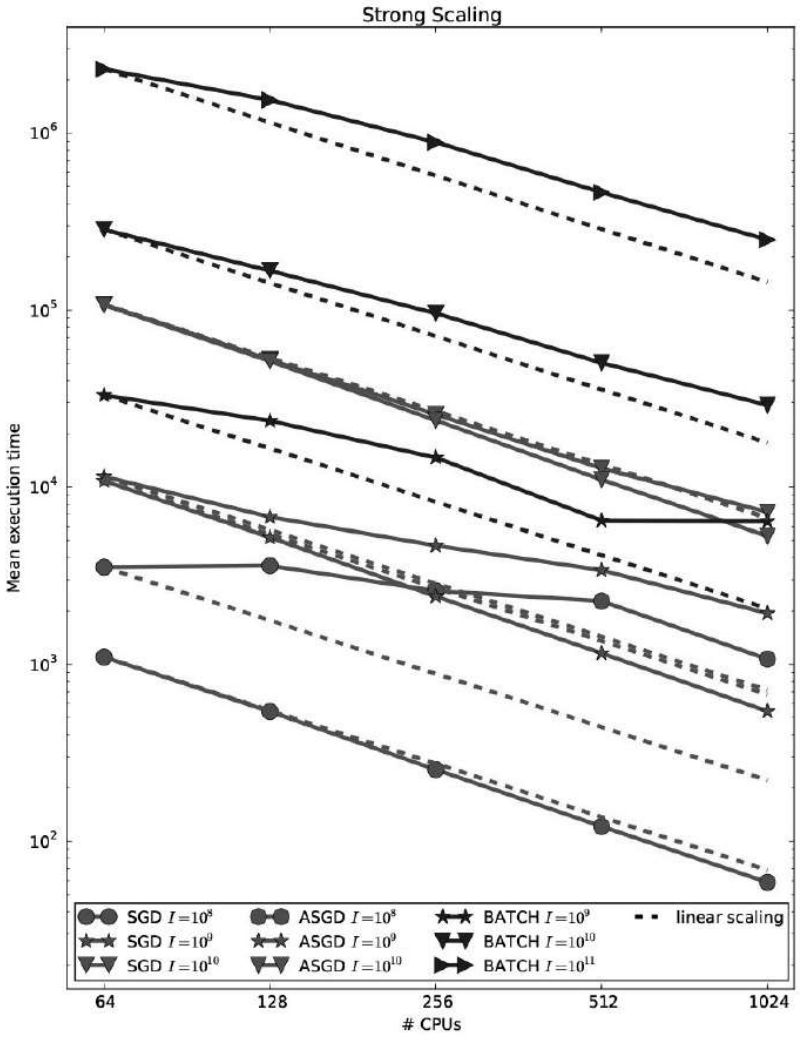


Figure 5: Results of a strong scaling experiment on the synthetic dataset with k=10, d=10 and ~1TB data samples for different numbers of iterations I. The related error rates are shown in figure 9.

Convergence Speed

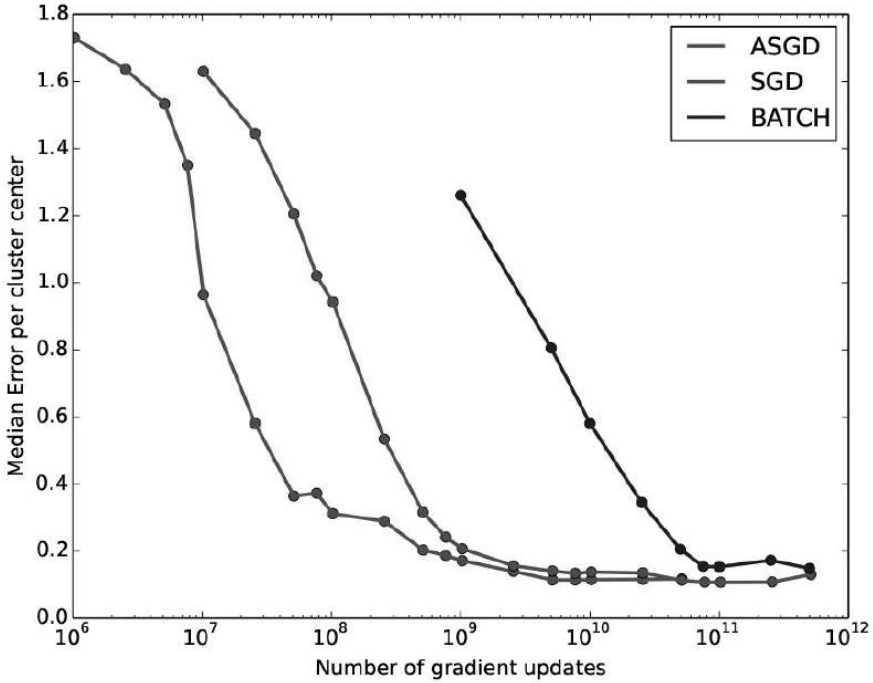


Figure 8: Convergence speed of different gradient descent methods used to solve K-Means clustering with $k = 100$ and $b = 500$ on a 10-dimensional target space parallelized over 1024 CPUs on a cluster. Our novel ASGD method outperforms communication free SGD [19] and MapReduce based BATCH [5] optimization by the order of magnitudes.

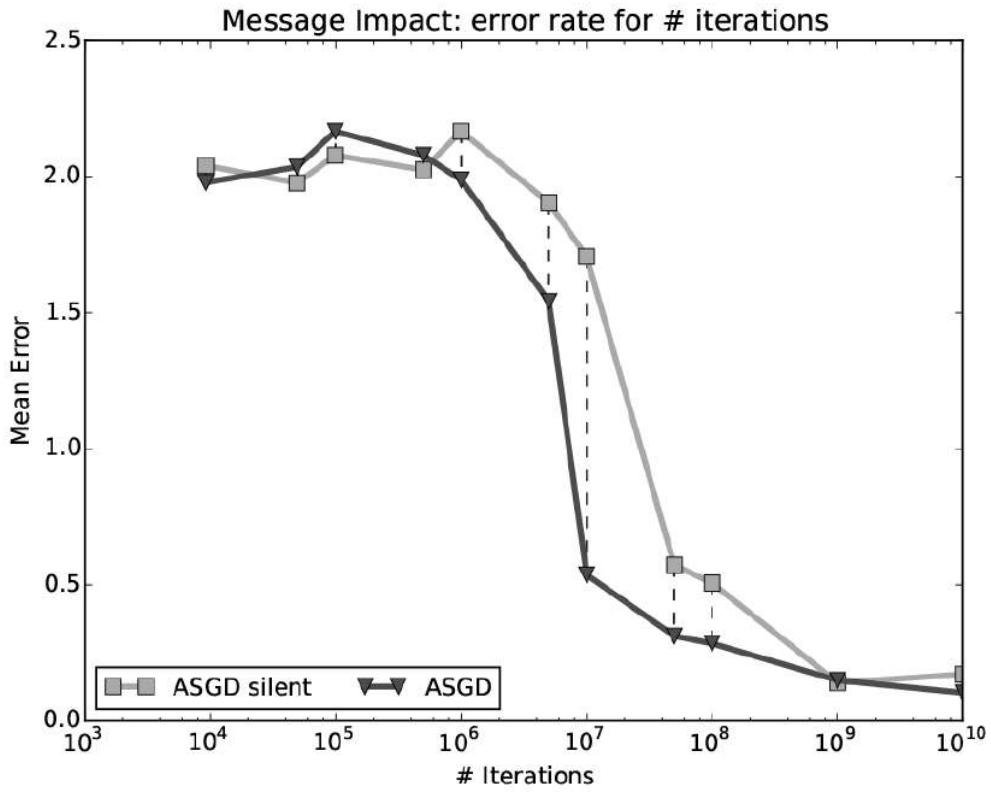


Figure 14: Convergence speed of ASGD optimization (synthetic dataset, $k = 10, d = 10$) with and without asynchronous communication (silent).