CSE 6220 Reading Notes

Jie Wu

Spring 2022

1 Week 0 Article

1.1 The Input/Output Complexity of Sorting and Related Problems

results: pp.1 LHS, three results

1. tight upper and lower bounds for the number of I/Os between internal memory and secondary storage required for five sorting-related problems (1) sorting (2) FFT (3) permutation networks (4) permuting (5) matrix permutation
2. give two optimal algorithms which are variants of merge sorting (with P-block look-ahead forecasting) and distribution sorting (with a median finding subroutine), for $P = 1$ the standard merge sorting algorithm is an optimal external sorting method
3. a more direct derivation of Hong and Kung's lower bound for the FFT for the special case $B = P = O(1)$

assumption: pp.1 LHS, secondary storage is modeled as a magnetic disk capable of transferring P blocks each containing B records in a single time unit; pp.2 LHS, each block transfer is allowed to access any contiguous group of B records on the disk

motivation: pp.1 LHS, much resources are consumed by external sorts, in which the file is too large to fit in internal memory and must reside in secondary storage, it is well documented that the bottleneck in external sorting is the time for I/O between internal memory and secondary storage

idea: (ref. TA's notes) where M is the memory size

$$N \log N \xrightarrow{B \text{ records per block}} (N/B)\log(N/B) \xrightarrow{P \text{ blocks simultaneously}} \frac{N/B}{P}\frac{\log(1+N/B)}{\log(1+M/B)}$$

theorem: pp.3 RHS, the average-case and worst-case number of I/Os required are (where O = worst case/upper bound vs. Ω = best case/lower bound)

$$\text{for } N\text{-input FFT diagraph:} \quad O\left(\frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}\right)$$

$$\text{for } N\text{-input permutation network:} \quad \Omega\left(\frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}\right)$$

$$\text{for permuting } N \text{ records:} \quad O\left(\min\left\{\frac{N}{P}, \frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}\right\}\right)$$

where the N/P bound reflects the fact that advance knowledge of the output permutation makes the problem of permuting easier than sorting; pp.6 RHS, and because permuting is a special case of sorting this lower bound also applies to sorting
pp.8 RHS, these bounds can be achieved by

variant of merge sort: besides the records themselves, we also place into each track $P-1$ "endmarkers", which are the key values of the last record (i.e. the largest key values) in the each of the next $P-1$ tracks of the run, using a generalization of the forecasting technique described in [5] we can then determine the P tracks that will expire next

variant of distribution sort: assume that we can compute the approximate partitioning elements using $O(N/PB)$ I/Os, then with $O(M/PB)$ additional I/Os we can input M records from disk into internal memory and partition them into the S bucket ranges, the records in each bucket range can be stored on disk in contiguous groups of B records each
the $O(M/PB)$ additional I/Os for partition can be achieved by picking the median record from each of the $\lceil n/M \rceil$ sets sorted internally in memory and find the median of the medians using the linear-time sequential algorithm developed in [2]

key facts & ideas: they are

- pp.3 LHS, the difference between permutation network and permuting

  permutation network: all $N!$ permutations can be generated by the same sequence of I/Os

  permuting: the particular I/Os performed may depend upon the desired permutation

- pp.3 RHS, key sorting consists of two steps: (1) stripping away the key values of the records (2) sorting the keys
- pp.9 RHS, three FFT diagraphs concatenated together form a permutation network, so it suffices to consider optimum algorithms for FFT
- pp.10 LHS, assuming p and q are powers of 2, matrix transposition is a special case of permuting; pp.8 LHS, the number of I/Os required to transpose a $p \times q$ matrix stored in row-major order is

$$\Omega\left(\frac{N}{PB}\frac{\log(\min\{M, 1+\min\{p,q\}, 1+N/B\})}{\log(1+M/B)}\right)$$

2 Week 2 Articles

2.1 Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers

1. quote only 32-bit performance results, not 64-bit results—it is hard to obtain impressive performance using 64-bit floating point arithmetic

2. present performance figures for an inner kernel and then represent these figures as the performance of the entire application—there is often a great deal of data movement and initialization that depresses overall performance rates

3. quietly employ assembly code and other low-level language constructs—it is often hard to obtain good performance from straightforward Fortran or C code that employs the usual parallel programming constructs, due to compiler weaknesses on many highly parallel computer systems

4. scale up the problem size with the number of processors but omit any mention of this fact—graphs of performance rates vs. the number of processors have a nasty habit of trailing off, this problem can easily be remedied by plotting the performance rates for problems whose sizes scale up with the number of processors

5. quote performance results projected to a full system—few labs can afford a full-scale parallel computer and the performance of a code on a scaled down system is often not very impressive, a straightforward solution to this dilemma is to project your performance results linearly to a full system without justifying the linear scaling

6. compare your results against scalar unoptimized code on Crays currently the world's dominant supercomputer—with a little tuning many applications run quite fast on Crays so do not insert vectorization directives, also Crays often run much slower with bank conflicts so access data with large power-of-two strides whenever possible, it is also important to avoid multitasking and autotasking on Crays

7. when direct run time comparisons are required compare with an old code on an obsolete system—compare your results with the performance of an obsolete code running on obsolete hardware with an obsolete compiler or less capable parallel system

8. if MFLOPS rates must be quoted base the operation count on the parallel implementation not on the best sequential implementation—parallel implementations often perform far more floating point operations than the best sequential implementation

9. quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar—it sounds great when you can claim that all processors are busy nearly 100% of the time even if what they are actually busy with is synchronization and communication overhead, or "fully linear" speedup simply by making sure that the single processor version runs sufficiently slowly, but sure not to use "sustained MFLOPS per dollar" i.e. actual delivered computational throughput per dollar

10. multilate the algorithm used in the parallel implementation to match the architecture—algorithmic changes are often necessary when we port applications to parallel computers, it is essential that you select algorithm which exhibit high MFLOPS performance rates without regard to fundamental efficiency

11. measure parallel run times on a dedicated system but measure conventional run times in a busy environment—make many runs on both systems and then publish the best time for the parallel system and the worst time for the conventional system

12. if all else fails show pretty pictures and animated videos and don't talk about performance—graphics often help deflect attention from the substantive technical issues

2.2 Introduction to Algorithms Chapter 27 Multithreaded Algorithms

- pp.773, dynamic multithreading allows programmers to specify parallelism in applications without worrying about communication protocols, load balancing, and other vagaries of static-thread programming

- pp.774, we can describe a multithreaded algorithm by adding to our pseudocode just three "concurrency" keywords: **parallel**, **spawn**, and **sync**, if we delete these concurrency keywords from the multithreaded pseudocode, the resulting text is serial pseudocode for the same problem, which we call the "serialization" of the multithreaded algorithm

2.2.1 The basics of dynamic multithreading

- pp.778, strand u spawning strand v differs from u calling v in that a spawn induces a horizontal continuation edge from u

- pp.779, for a computation DAG in which each strand takes unit time, the **work** is just the number of vertices in the DAG, and the **span** equals the number of vertices on a longest or critical path in the DAG, the **work law** says

$$T_P \geq T_1/P$$

  the **span law** says

$$T_P \geq T_\infty$$

- pp.780, when the speedup is linear in the number of processors, that is when $T_1/T_P = \Theta(P)$ the computation exhibits linear speedup, and when $T_1/T_P = P$ we have perfect linear speedup, the ratio T_1/T_∞ of the work to the span gives the **parallelism** of the multithreaded computation, we define the parallel **slackness** of a multithreaded computation executed on an ideal parallel computer with P processors to be the ratio $(T_1/T_\infty)/P$, it is the factor by which the parallelism of the computation exceeds the number of processors in the machine

- pp.782, **greedy schedulers** assign as many strands to processors as possible in each time step, if at least P strands are ready to execute during a time step we say that the step is a **complete step**, otherwise fewer than P strands are ready to execute in which case we say that the step is an **incomplete step**, on an ideal parallel computer with P processors, a greedy scheduler executes a multithreaded computation with work T_1 and span T_∞ in time

$$T_P \leq T_1/P + T_\infty \leq 2 \cdot \max(T_1/P, T_\infty)$$

  let T_P^* be the running time produced by an optimal scheduler on a machine with P processors, by the work law and span law we have

$$T_P^* \geq \max(T_1/P, T_\infty) \implies T_P \leq 2T_P^*$$

  if $P \ll T_1/T_\infty$ we have $T_P \approx T_1/P$

- pp.776, for Fibonacci the work is (where $\phi = (1+\sqrt{5})/2$)

$$\left.\begin{aligned} T(n) = T(n-1) + T(n-2) + \Theta(1) \\ T(n) \leq aF_n - b \end{aligned}\right\} \implies T(n) = \Theta(F_n) = \Theta(\phi^n)$$

pp.784, the span is

$$T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) = T_\infty(n-1) + \Theta(1) \implies T_\infty(n) = \Theta(n)$$

thus the parallelism is $T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$

- pp.787, the overhead of recursive spawning does increase the work of a parallel loop compared with that of its serialization but not asymptotically, thus we can amortize the overhead of recursive spawning against the work of the iterations, contributing at most a constant factor to the overall work
  as a practical matter dynamic-multithreading concurrency platforms sometimes **coarsen** the leaves of the recursion by executing several iterations in a single leaf, thereby reducing the overhead of recursive spawning, this reduced overhead comes at the expense of also reducing the parallelism

- pp.788, a **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions perform a write; pp.789, we shall ensure that strands that operate in parallel are independent, thus

  parallel for: all the iterations should be independent

  spawn + sync: the code of the spawned child should be independent of the code of the parent

2.2.2 Multithreaded matrix multiplication

parallel for matrix multiplication: pp.793, the work is the same as square matrix multiplication

$$T_1(n) = \Theta(n^3)$$

using down the tree of recursion for the two *parallel for* loops over i and j and then executes all n iterations of the ordinary *for* loop over k i.e. $c_{ij} = c_{ij} + a_{ik}b_{kj}$ results in a total span of

$$T_\infty(n) = \Theta(\log n) + \Theta(\log n) + \Theta(n) = \Theta(n)$$

thus the parallelism is $\Theta(n^2)$

divide-and-conquer submatrix multiplication: pp.795, the work is (where $\Theta(n^2)$ is for adding two $n \times n$ matrices)

$$M_1(n) = 8M_1(n/2) + \Theta(n^2) \implies M_1(n) = \Theta(n^3)$$

by case 1 of the master theorem
the span is (where $\Theta(\log n)$ is for the doubly nested *parallel for* loop $c_{ij} = c_{ij} + t_{ij}$)

$$M_\infty(n) = M_\infty(n/2) + \Theta(\log n) \implies M_\infty(n) = \Theta(\log^2 n)$$

by the solution of Exercise 4.6-2, thus the parallelism is $M_1(n)/M_\infty(n) = \Theta(n^3/\log^2 n)$

2.2.3 Multithreaded merge sort

serial merge: pp.797, because merge is serial both its work and its span are $\Theta(n)$, the following recurrence characterizes the work

$$MS'_1(n) = 2MS'_1(n/2) + \Theta(n) \implies MS'_1(n) = \Theta(n \log n)$$

which is the same as the serial running time of merge sort
since two recursive calls of merge sort can run in parallel, the span is given by the recurrence

$$MS'_\infty(n) = MS'_\infty(n/2) + \Theta(n) \implies MS'_\infty(n) = \Theta(n)$$

thus the parallelism of merge sort is $MS'_1(n)/MS'_\infty(n) = \Theta(\log n)$

parallel merge: pp.800, to expedite the serial merge, compute the midpoint q_1 of $T[p_1..r_1]$ and find the point q_2 in $T[p_2..r_2]$ such that all the elements in $T[p_2..q_2-1]$ are less than $T[q_1]$ and all the elements in $T[q_2..p_2]$ are at least as large as $T[q_1]$, so that we can merge $T[p_1..q_1-1]$ and $T[p_2..q_2-1]$ at the same time we merge $T[q_1+1..r_1]$ and $T[q_2..r_2]$, in the worst case all the n_2 elements $T[q_2..r_2]$ are less than q_1 so that the number of elements involved in the merge is $\lfloor n_1/2 \rfloor + n_2 \le (n_1+n_2)/2 + n_2/2 = 3n/4$, thus the worst-case span satisfies the following recurrence (where $\Theta(\log n)$ is the cost of binary searching q_1)

$$PM_\infty(n) = PM_\infty(3n/4) + \Theta(\log n) \implies PM_\infty(n) = \Theta(\log^2 n)$$

by the solution of Exercise 4.6-2
pp.801, the work of parallel merge is $PM_1(n) = \Theta(n)$, because on the one hand $PM_1(n) = \Omega(n)$ since each of the n elements must be copied, on the other hand $PM_1(n)$ satisfies

$$PM_1(n) = PM_1(\alpha n) + PM_1((1-\alpha)n) + O(\log n) = O(n)$$

where $1/4 \le \alpha \le 3/4$ as in the worst span case above, this recursive relation has the solution $PM_1(n) = O(n)$ because choosing c_2 large enough we have $PM_1(n) \le c_1 n - c_2 \log n$ satisfies the recurrence, since $PM_1(n)$ is both $\Omega(n)$ and $O(n)$ we have $PM_1(n) = \Theta(n)$, therefore the parallelism of parallel merge is $PM_1(n)/PM_\infty(n) = \Theta(n/\log^2 n)$

parallel merge sort: pp.803, the work is given by the recurrence

$$PMS_1(n) = 2PMS_1(n/2) + PM_1(n) = 2PMS_1(n/2) + \Theta(n) \implies PMS_1(n) = \Theta(n \log n)$$

by case 2 of the master theorem
similarly the span is given by the recurrence

$$PMS_\infty(n) = PMS_\infty(n/2) + PM_\infty(n) = PMS_\infty(n/2) + \Theta(\log^2 n) \implies PMS_\infty(n) = \Theta(\log^3 n)$$

by the solution of Exercise 4.6-2, thus the parallelism is $PMS_1(n)/PMS_\infty(n) = \Theta(n/\log^2 n)$
pp.804, a good implementation in practice would sacrifice some parallelism by coarsening the base case in order to reduce the constant hidden by the asymptotic notation, the straightforward way to coarsen the base case is to switch to an ordinary serial sort

3 Week 3 Article

3.1 Introduction to Parallel Computing Chapter 7 Programming Shared Address Space Platforms

pp.1, process based models assume that all data associated with a process is private by default, in contrast lightweight processes and threads assume that all memory is global, as a result this is the preferred model for parallel programming

3.1.1 Why Threads?

pp.2, threaded programming models offer significant advantages over message-passing programming models

software portability: threaded applications can be developed on serial machines and run on parallel machines without any changes

latency hiding: one of the major overheads in programs (both serial and parallel) is the access latency for memory access, I/O, and communication, by allowing multiple threads to execute on the same processor threaded APIs enable this latency to be hidden

scheduling and load balancing: threaded APIs allow the programmer to support system-level dynamic mapping of tasks to processors with a view to minimize idling overheads, by providing this support at the system level threaded APIs rid the programmer of the burden of explicit scheduling and load balancing

ease of programming, widespread use: threaded programs are significantly easier to write than corresponding program using message passing APIs

3.1.2 Thread Basics: Creation and Termination

pp.7, this situation in which two threads "falsely" share data because it happens to be on the same cache line is called false sharing
pp.7, it is in fact possible to use this simple example to estimate the cache line size of the system—by changing the size of the second dimension we can force entries in the first column of the array to lie on different cache lines (since arrays in C are stored row-major), and spacing the entries out pushes them into different cache lines reduces the false sharing overhead

3.1.3 Synchronization Primitives in Pthreads

- pp.13, locks represent serialization points since critical sections must be executed by threads one after the other, encapsulating large segments of the program within locks can therefore lead to significant performance degradation

- pp.13, it is often possible to reduce the idling overhead associated with locks using an alternate function *pthread_mutex_trylock*, this function attempts a lock on *mutex_lock*, if the lock is successful the function returns a zero, if it is already locked by another thread instead of blocking the thread execution it returns a value EBUSY, this allows the thread to do other work, furthermore it *pthread_mutex_trylock* is typically much faster than *pthread_mutex_lock*

on typical systems since it does not have to deal with queues associated with locks for multiple threads waiting on the lock, and the number of locking operations is also reduced

- pp.16, in addition to blocking the thread the *pthread_cond_wait* function releases the lock on mutex, when the thread is released on a signal it waits to reacquire the lock on mutex before resuming execution; pp.18, it is a good practice to check for the condition in a loop because the thread might be woken up due to other reasons

- pp.18, when a thread performs a condition wait it takes itself off the runnable list, consequently it does not use any CPU cycles until it is woken up, this is in contrast to a mutex lock which consumes CPU cycles as it polls for the lock

3.1.4 Controlling Thread and Synchronization Attributes

pp.21, a recursive mutex allows a single thread to lock a mutex multiple times, each time a thread locks the mutex a lock counter is incremented, each unlock decrements the counter, for any other thread to be able to successfully lock a recursive mutex the lock counter must be zero, a recursive mutex is useful when a thread function needs to call itself recursively

3.1.5 OpenMP: a Standard for Directive Based Parallel Programming

pp.31, while standardization and support for threaded APIs has come a long way their use is still predominantly restricted to system programmers as opposed to application programmers, one of the reasons for this is that APIs such as pthreads are considered to be low-level primitives, conventional wisdom indicates that a large class of applications can be efficiently supported by higher level constructs (or directives) which rid programmer of the mechanics of manipulating threads, OpenMP directives provide support for concurrency, synchronization, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization

firstprivate vs. private: firstprivate is similar to private except the values of variables on entering the threads are initialized to corresponding values before the parallel directive

default (shared) vs. default (none): default (shared) implies that by default a variable is shared by all the threads vs. default (none) implies that the state of each variable used in a thread must be explicitly specified

reduction: specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit

3.1.5.1 Specifying concurrent tasks in OpenMP

lastprivate: deals with how multiple local copies of a variable are written back into a single copy at the end of the parallel *for* loop, since it is sometimes desired that the last iteration of the *for* loop update the value of a variable

schedule(static) vs. schedule(dynamic) vs. schedule(guided): static scheduling splits the iteration space into equal chunk size, when no chunk size is specified the iteration space is split into as many chunks as there are threads and one chunk is assigned to each thread
dynamic scheduling partitions the iteration space into chunks and assigns chunks to threads

as they become idle, when no chunk size is specified it defaults to a single iteration per chunk
guided scheduling reduces the chunk size exponentially as each chunk is dispatched to a thread, the specified chunk size refers to the smallest chunk that should be dispatched, therefore when the number of iterations left is less than the specified chunk size the entire set of iterations is dispatched at once, the chunk size defaults to one if none is specified

pp.37, the *for* directive places additional restrictions on the *for* loop that follows

- it must not have a break statement
- the loop control variable must be an integer
- the initialization expression of the *for* loop must be an integer assignment
- the logical expression must be one of $<$,$\leq$,$>$,or $\geq$
- the increment expression must have integer increments or decrements only

nowait: used with a *for* directive to indicate that the threads can proceed to the next statement without waiting for all other threads to complete the *for* loop execution

section: the *sections* directive assigns the structured block corresponding to each section to one thread, at the end of execution of the assigned section the threads synchronize (unless the nowait clause is used), note that it is illegal to branch in and out of section blocks

pp.41, instead of nesting three *for* directives inside a single *parallel* directive we have used three *parallel for* directives, this is because OpenMP does not allow *for*, *sections*, and *single* directives that bind to the same *parallel* directive to be nested, to generate a new set of threads nested parallelism must be enabled using the *OMP_NESTED* environment variable, the default state of this environment variable is FALSE meaning the inner *parallel* region is serialized

3.1.5.2 Synchronization constructs in OpenMP

barrier: on encountering this directive all threads in a team wait until others have caught up and then release, when used with nested *parallel* directives the *barrier* directive binds to the closest *parallel* directive

single: on encountering the *single* block the first thread enters the block, all the other threads proceed to the end of the block, if the *nowait* clause has been specified at the end of the block, then the other threads proceed, otherwise they wait at the end of the *single* block for the thread to finish executing the block, this directive is useful for computing global data as well as performing I/O

master: a specialization of the *single* directive in which only the master thread executes the structured block, in contrast to the *single* directive there is no implicit barrier associated with the *master* directive

critical: OpenMP provides a *critical* directive for implementing critical regions that must be executed serially one thread at a time, the optional name field can be used to identify a critical region, which allows different threads to execute different code while being protected from each other, if no name is specified the critical section maps to a default name that is the same for all unnamed critical sections, the names of critical sections are global across the program
as is the case with pthreads we must reduce the size of the critical sections as much as possible to get good performance

atomic: it is important to note that the *atomic* directive only atomizes the load and store of the scalar variable, the evaluation of the expression is not atomic thus the expression cannot contain the updated variable itself, all *atomic* directives can be replaced by *critical* directives provided they have the same name, however the availability of atomic hardware instructions may optimize the performance of the program compared to translation to *critical* directives

ordered: represents an ordered serialization point in the program e.g. cumul_sum[i] = cumul_sum[i-1] + list[i], only a single thread can enter an ordered block, a single *for* directive is constrained to have only one *ordered* block in it

flush: provides a memory fence by forcing a variable to be written to or read from the memory system, all write operations to shared variables must be committed to memory at a flush and all references to shared variables after a fence must be satisfied from the memory, since private variables are relevant only to a single thread the *flush* directive applies only to shared variables, the default is that all shared variables are flushed, several OpenMP directives have an implicit *flush*

3.1.5.3 Data handling in OpenMP

firstprivate: if a thread repeatedly reads a variable that has been initialized earlier in the program, it is beneficial to make a copy of the variable and inherit the value at the time of thread creation, this way when a thread is scheduled on the processor, the data can reside at the same processor (in its cache if possible) and accesses will not result in interprocessor communication

threadprivate: objects locally available to a thread in such a way that these objects persist through parallel and serial blocks provided dynamic adjustment of the number of threads is disabled and the number of threads remains the same, in contrast to *private* variables these variables are useful for maintaining persistent objects across parallel regions, which would otherwise have to be copied into the master thread's data space and reinitialized at the next parallel block, *copyin* provides a mechanism for assigning the same value to *threadprivate* variables across all threads in a parallel region

3.1.5.4 OpenMP library functions

pthread	OpenMP
pthread_mutex_init	omp_init_lock
pthread_mutex_destroy	omp_destroy_lock
pthread_mutex_lock	omp_set_lock
pthread_mutex_unlock	omp_unset_lock
pthread_mutex_trylock	omp_test_lock

3.1.5.5 Environment variables in OpenMP

pp.49, setenv OMP_SCHEDULE "static,4" specifies that all *for* directives use static scheduling with a chunk size of 4
pp.49, other environment variables are OMP_NUM_THREADS, OMP_DYNAMIC, OMP_NESTED

3.1.5.6 Explicit threads versus OpenMP based programming

pp.50, the drawbacks to using directives are

1. an artifact of explicit threading is that data exchange is more apparent, this helps in alleviating some of the overheads from data movement, false sharing, and contention
2. explicit threading also provides a richer API and increased flexibility for building composite synchronization operations
3. since explicit threading is used more widely than OpenMP, tools and support for pthreads programs is easier to find

4 Week 4 Articles

4.1 Prefix Sums and Their Applications

Implementation: pp.38, scan = sum of all the previous elements including the element itself vs. prescan = sum of all the previous elements excluding the element itself
pp.39, if we assume a fixed number of processors p with $n > p$, then each processor can sum an $\lceil n/p \rceil$ section of the vector to generate a processor sum, the tree technique can then be used to reduce the processor sums to total sum, thus the total time required is

$$T_R(n,p) = \lceil n/p \rceil + \lceil \log p \rceil = O(n/p + \log p)$$

when $n/p \geq \log p$ the complexity is $O(n/p)$
pp.43, to implement the prescan on an PRAM, the partial sums at each vertex must be kept during the up-sweep so they can be used during the down-sweep, thus the time complexity of the algorithm is

$$T_S(n,p) = 2\left(\lceil n/p \rceil + \lceil \log p \rceil\right) = O(n/p + \log p)$$

whic is the same order as the reduce operation ($\Rightarrow$ but a factor of 2 more!)

line-of-sight and radix-sort: pp.46, in a radix sort the split operation packs the keys with a 0 in the corresponding bit to the bottom of a vector, and packs the keys with a 1 in the bit to the top of the same vector, it requires two scan operations

- to determine the new indices for elements with a 0 in the bit, invert the flags and execute a prescan with integer addition
- to determine the new indices for elements with a 1 in the bit, execute a +-scan in reverse order (starting at the top of the vector) and subtract the results from the length of the vector

if we assume that n keys are each $O(\log n)$ bits long, then the overall radix sort runs in time

$$O\left((n/p + \log p)\log n\right) = O\left(n/p \log n + \log p \log n\right)$$

recurrence equations: pp.48, the scan operation is the special case of a recurrence of the form $x_i = x_{i-1} \oplus a_i$, to convert $x_i = (x_{i-1} \otimes a_i) \oplus b_i$ into it, define the set of pairs $c_i = [a_i, b_i]$ and the ordered set $s_i = [y_i, x_i]$ where y_i obeys the recurrence $y_i = y_{i-1} \odot a_i$ so that $s_i = s_{i-1} \cdot c_i$

segmented scan: pp.51, a scan can be broken into segments with flags so that the scan starts again at each segment boundary, each of these scans takes two vectors of values: a data vector and a flag vector, the segmented scan operations present a convenient way to execute a scan independently over many sets of values, this property can be used to execute a parallel quicksort; pp.52, the basic intuition is to keep each subset in its own segment, and to pick pivot values and split the keys independently within each segment; pp.53, if we select pivots randomly within each segment, quicksort is expected to complete in $O(\log n)$ iterations, and therefore has an expected running time of $O(\log n \cdot T_S(n, p))$
pp.53, the technique of recursively breaking segments into subsegments and operating independently within each segment can be used for many other divide-and-conquer algorithms such as mergesort

4.2 Designing Practical Efficient Algorithms for Symmetric Multiprocessors

abstract: pp.37, we present a computational model for designing efficient algorithms for symmetric multiprocessors (SMP), we then use this model to create efficient solutions to two widely different types of problems—linked list prefix computations and generalized sorting, in spite of the fact that the processors must compete for access to main memory, both algorithms still yielded scalable performance

emphasis on memory access: pp.38, the rapid progress in microprocessor speed has left main memory access as the primary limitation to SMP performance, since memory is the bottleneck, simply increasing the number of processors will not necessarily yield better performance, this has at least two implications for the algorithm designer

1. any parallel algorithm must be competitive with its sequential counterpart with as little as one processor in order to be relevant
2. for the parallel algorithm to scale with the number of processors, it must be designed with careful attention to minimizing the number and type of main memory accesses

algorithm complexity measure: pp.40, we measure the overall complexity of an algorithm by the triplet (M_A, M_E, T_C) where

M_A**:** the maximum number of accesses made by any processor to main memory, it is simply a measure of the number of non-contiguous main memory accesses where each such access may involve an arbitrary sized contiguous block of data

M_E**:** the maximum amount of data exchanged by any processor with main memory

T_C**:** an upper bound on the local computational complexity of any of the processors

while we report T_C using the customary asymptotic notation, we report M_A and M_E as approximations of the actual values (⇒ ignore the lower power terms in the polynomial), we distinguish between memory access cost and computational cost in this fashion because of the dominant expense of memory access on this architecture
pp.41, in practice it is often possible to focus on either M_A or M_E when examining the cost of algorithmic alternatives, e.g. we observed when comparing prefix computation algorithms that the number of contiguous and non-contiguous memory accesses were always of the same asymptotic order, and therefore we only report M_A which describes only the much more expensive non-contiguous accesses

4.2.1 Conventional prefix algorithms

4.2.1.1 sequential algorithm

1. visit each list element and label its successor ⇒ at most n non-contiguous accesses
2. find the one not labeled as having a predecessor which is the head ⇒ one non-contiguous memory access
3. beginning at the head traverse the elements by following the successor pointers and take prefix sum ⇒ at most n non-contiguous memory accesses

total $2n$ non-contiguous memory accesses and $O(n)$ computation time

4.2.1.2 optimal sequential algorithm

1. compute the sum Z of the successor indices by visiting each list element, the index of head of the list is $h = n(n-1)/2 - Z$ ⇒ head index is missing in the set of successor indices
2. same as the sequential algorithm

total n non-contiguous memory accesses and $O(n)$ computation time

4.2.1.3 parallel algorithm overview

Wyllie: $n \log n$ non-contiguous accesses

Vishkin: $5n$ non-contiguous accesses

Anderson and Miller: $4n$ non-contiguous accesses

Reid-Miller and Blelloch: $2n$ non-contiguous accesses

none match the optimal sequential algorithm

4.2.2 Helman-JaJa algorithm idea

pp.43, partition the input list into s sublists by randomly choosing exactly one splitter (list head is the pre-designated one) from each memory block of $n/(s-1)$ elements where $s = \Omega(p \log n)$

***List*:** two fields

successor: the integer index of the successor of the element

prefix_data: initially holds the input value for the prefix operation, final output is the properly computed prefix value

***Sublist*:** four fields

head: the *List* index of the splitters (*List* head as the first splitter)

scratch: the value of the *List* successor of the splitters

prefix_data: the *List* prefix_data of the splitters (0 for *List* head)

successor: the sublist index of the next sublist

4.2.3 Helman-JaJa algorithm steps

1. each processor P_i visits $i(n/p)$ through $(i+1)(n/p)-1$ *List* elements to compute the sum of the successor indices which is stored in location i of the array Z (the terminal element successor index is set to equal $-s$ and not included in the sum)
   during the visit the value of in the successor field is saved to an identically indexed location in the array *Succ*

   M_A: 3 (*List*, Z, *Succ*)

   M_E: $2n/p$ (sum, save)

   T_C: $O(n/p)$

2. one processor P_0 computes the sum T of the array Z, from which the *List* head is determined by $h = n(n-1)/2 - T$

   M_A: 1 (Z)

   M_E: p (one successor index sum per processor)

   T_C: $O(p)$ (Z)

3. each processor P_i is assigned s/p of the s sublists indexed by $j = i(s/p)$ through $(i+1)(s/p)-1$ which are defined by randomly choosing *List* index $(j-1)[n/(s-1)]$ through $j[n/(s-1)]-1$
   for the first elements (the splitters)

   - the *List* index of the splitters is saved as *Sublists*[j].head
   - the value of the *List* successor of the splitters is saved to *Sublists*[j].scratch, after which the value of the *List* successor of the splitters set to equal $-j$

   M_A: s/p (number of splitters per processor)

M_E: $2s/p$ (*List* index and *List* successor of the splitters)

T_C: $O(s/p)$ (random number generation)

4. for each assigned sublist processor P_i traverses the elements, update their value of the *List* successor to be $-j$ and compute their *List* prefix_data
for the last elements (the splitters)

   - the value of the *Sublist* successor is set to equal the sublist index of the next sublist
   - the *Sublist* prefix_data is set of equal the *List* prefix_data

   M_A: at most $\alpha(s)(n/p)$ with high probability (s/p splitters per processor)

   M_E: $\alpha(s)(n/p)$, ignore the cost of writing last element successor and prefix_data as $s \ll n$

   T_C: $O(n/p)$ with high probability (prefix_data computation per processor)

5. one processor P_0 traverses *Sublist* to computer the *Sublist* prefix_data

   M_A: s (number of splitters)

   M_E: s (*Sublist* prefix_data)

   T_C: $O(s)$ (sequential)

6. each processor P_i visits $i(n/p)$ through $(i+1)(n/p)-1$ *List* elements to sum their *List* prefix_data and the *Sublist* prefix_data of the first element/head of the sublists they belong to (labeled by their value of the *List* successor), after than the values of their *List* successor are restored by copying from the *Succ* array

   M_A: s (assuming *Sublist* fits into the cache) & $s+1$ (copy back from *Succ*)

   M_E: $2n/p$ (head prefix_data, copy back)

   T_C: $O(n/p)$ (sum per processor)

overall complexity is

	step 1	step 2	step 3	step 4	step 5	step 6	total
M_A	3	1	s/p	$\alpha(s)(n/p)$	s	$2s+1$	$\alpha(s)(n/p)$
M_E	$2n/p$	p	$2s/p$	$\alpha(s)(n/p)$	s	$2n/p$	$[\alpha(s)+4](n/p)$
T_C	$O(n/p)$	$O(p)$	$O(s/p)$	$O(n/p)$	$O(s)$	$O(n/p)$	$O(n/p)$

5 Week 5 Article

5.1 Parallel Tree Contraction and Its Application

5.1.1 Introduction

pp.1 LHS, we give a bottom-up algorithm to handle trees, that is all modifications to the tree are done locally, we call this bottom-up approach **contract**
we shall use the PRAM model which consists of a collection of processors each of which can read and write in a common RAM, in unit time they are allowed concurrent reads and concurrent writes (CRCW)

5.1.2 The RAKE and COMPRESS Operations

RAKE: pp.2 RHS, the operation of removing all leaves from tree T, RAKE may need to be applied linear number of times to a highly unbalanced tree to reduce T to a single node, we can circumvent this problem by adding one more operation

chain: pp.2 RHS, a sequence of nodes $v_1, \ldots, v_k$ is a **chain** if (1) v_{i+1} is the only child of v_i for $1 \leq i < k$ and (2) v_k has exactly one child and that child is not a leaf, in one parallel step we **compress** a chain by identifying v_i with v_{i+1} for i odd and $1 \leq i < k$

COMPRESS: pp.2 RHS, the operation on T which contracts all maximal chains of T in one step, maximal chains of length one are not affected by COMPRESS

CONTRACT: pp.2 RHS, the simultaneous application of RAKE and COMPRESS to the entire tree, the CONTRACT operation need only be executed $O(\log n)$ times, more precisely $\lceil \log 5/4n \rceil$ (ref. theorem 1), to reduce T to its root, this can be shown by partitioning the vertices V of a tree $T = (V, E)$ into two sets

RAKE: RAKE$(V) = Ra = V_0 \cup V_2 \cup C_0 \cup C_2 \cup GC_0$

COM: COMPRESS$(V) = Com = V - Ra$

where V_0 = leaves of T, V_1 = vertices with only one child, V_2 = vertices with 2 or more children, C_0, C_1 and C_2 are the vertices in V_1 whose child is in V_0, V_1 and V_2, GC_0, GC_1 and GC_2 are the vertices in C_1 whose child is in V_0, V_1 and V_2, the claim is that the RAKE operation will decrease the number of vertices by a factor of 4/5 and the COMPRESS operation will decrease it by a factor of 1/2

5.1.3 Dynamic Tree Contraction

pp.3 LHS, two implementations of COMPRESS

deterministic: need $O(n)$ processors to achieve $O(\log n)$ time

randomized: need $O(n/\log n)$ processors to achieve $O(\log n)$ time

5.2 Professor's Notes on the Paper

5.2.1 Rake and Compress

pp.1, **expression evaluation** = the tree represents an expression, every node is an operation or function, and subtrees are the argument expressions, **compress** = pointer jumping and **rake** = leave removal can be applied in parallel to disjoint parts of a tree—compress produces leaves for rake & rake produces linear lists for compress

5.2.2 Basic tree contraction CREW PRAM algorithm

pp.2, after $O(\log 4/3n)$ applications of basic contract (rake + compress) to n-node tree T it is reduced to a root, if rake and compress take $O(1)$ time then the parallel time with $O(n)$ processors is $O(\log n)$

5.2.3 Binary expression tree evaluation

pp.2, if T represents an arithmetic expression then internal nodes represent operators and leaves contain constant input values, every edge $\langle v, P[v]\rangle$ is labeled with a linear function $f_v(x) = a_v x + b$ where a_v, b_v are integer constants, initially $f_v(x) = x$ for all $v \in T$, $v \neq$ root, the rake of a leaf v means placing $f_v(val[v])$ on the edge $\langle v, P[v]\rangle$
pp.2, consider an internal node v with $op[v]$ whose right son r supplied its constant value $val[r]$ by placing $c_r = f_r(val[r]) = a_r val[r] + b_r$ on edge $\langle r, v\rangle$, assume that the left child l does not know its value i.e. its subtree has not been evaluated yet, then node v can be compressed i.e. jumped over, and the linear function becomes $f'_l(x) = a'_l x + b'_l = a_v[(a_l x + b_l)op[v]c_r] + b_v$

5.2.4 Work-optimal EREW PRAM tree contraction

pp.3, the previous algorithm has two drawbacks

not work-optimal: the number of operations required is $n \log n$ in contrast to $O(n)$ in sequential evaluation, because whenever a compress is performed we get two chains, but at most one of them is essential for the evaluation of the accumulated value in the root, the other chain is nonessential for the total value but processors keep contracting it

requires concurrent-read PRAM: a head of a chain—a node with two unevaluated children among which one is the last node of a linked list of nodes that have one child evaluated—becomes the parent of all the nodes from the linked list thanks to compressing, the head cannot be jumped over until its second child submits its value (i.e. marked), all these waiting nodes will attempt to read $P[P[v]]$ once set

pp.4, both difficulties can be overcome if the contraction will not produce linear chains at all, a chain is produced when the tree contains a binary subtree where each internal vertex has one child that is a leaf and one that is not, if every individual RAKE operation is followed immediately by a COMPRESS of its sibling chains cannot be formed, this combined operation is called SHUNT, there are several situations in which parallel execution of SHUNT operations could cause problems

- SHUNT is not defined for children of the root, because COMPRESS cannot be applied to the root
- SHUNT cannot be performed on two siblings simultaneously, because not only would it require concurrent-read PRAM but it could lead to a nondeterministic result due to parallel racing
- SHUNT cannot be performed on two adjacent leaves in left-to-right ordering, because it would disconnect the tree
- SHUNT cannot be applied to consecutive left and right odd-numbered leaves, not only does it disconnect the tree but it is nondeterministic

pp.5, to implement this kind of contraction we need to order leaves of tree, this is equivalent to numbering the leaves in depth-first order, hence it is a similar problem as pre-order numbering except that the internal nodes do not count, call this procedure **LR_numbering**
pp.6, SHUNT has the following two properties

1. it runs correctly on EREW PRAM, because it follows from the definition of SHUNT that no collision can occur for two nonconsecutive left leaves of T

2. using $n/\log n$ processors the runtime of SHUNT is $O(\log n)$, because assigning $\log n/2$ leaves to each processor, since RAKE eliminates $1/2$ of current leaves, the total number of SHUNT operations is at most

$$\frac{\log n}{2^1} + \frac{\log n}{2^2} + \cdots + \frac{\log n}{2^{\log\log n}} + 1 + \cdots + 1 \leq 2\log n$$

6 Week 6 Article

6.1 A Work-Efficient Parallel Breadth-First Search Algorithm (Optional)

- there are actually two data race conditions (1) assigning distance (2) inserting an element to a bag, the latter would lead to nondeterministic workload, to address this the authors introduce an additional data structure called reducer

- the authors also adopt a special "work-stealing" scheduler to achieve work balance across the bits of the spine

7 Week 9 Article

7.1 Introduction to Parallel Computing Chapter 6 Programming Using the Message-Passing Paradigm

7.1.1 Principles of Message-Passing Programming

pp.1, there are two key attributes that characterize the message-passing programming paradigm

it assumes a partitioned address space: pp.1, there are two immediate implications of a partitioned address space

- data must be explicitly partitioned and placed, which adds complexity but encourages locality
- the process that has the data must participate in the interaction even if it has no logical connection to the events, as a consequence the code complexity of dynamic programming can be very high

it supports only explicit parallelization: pp.1, the programmer is responsible for analyzing the underlying serial algorithm/application and identifying ways by which he or she can decompose the computations and extract concurrency

pp.1, asynchronous execution makes it possible to implement any parallel algorithm, but can have non-deterministic behavior due to race conditions, loosely synchronous programs—tasks synchronize to perform interactions but execute completely asynchronously between interactions—are a good compromise
pp.2, to make the job of writing parallel programs scalable, most message-passing programs are

written using the single program multiple data (SPMD) approach, in which the code executed by different processes is identical except for a small number of processes

7.1.2 The Building Blocks: Send and Receive Operations

pp.2, most message passing platforms have additional hardware support for sending and receiving messages such as

direct memory access (DMA): allows copying data from one memory location to another without CPU support

network interface: allows the transfer of messages from buffer memory to desired location without CPU intervention

7.1.2.1 Blocking message passing operations

pp.3, the sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently, there are two mechanisms by which this can be achieved

blocking non-buffered send/receive: pp.3, the send operation does not return until the matching receive has been encountered at the receiving process, typically this process involves a handshake between the sending and receiving processes, since there are no buffers used at either sending or receiving ends, this is also referred to as a non-buffered blocking operation
pp.3, nevertheless there is considerable idling at the sending and receiving process if the send and receive are not posted at roughly the same time, this idling overhead is one of the major drawbacks of this protocol
pp.4, another major drawback of this protocol is that, deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits of the nature outlined

blocking buffered send/receive: pp.4, on encountering a send operation the sender simply copies the data into the designated buffer and returns after the copy operation has been complete, at the receiving end the data is copied into a buffer at the receiver as well, when the receiving process encounters a receive operation, it checks to see if the message is available in its receive buffer, if so the data is copied into the target location
pp.4, sometimes machines do not have such communication software to support the buffers, in this case some of the overhead can be saved by buffering only on one side through e.g. interrupts, the buffering can similarly be done only at the sender and the receiver initiates a transfer by interrupting the sender
pp.5, buffered protocols alleviate idling overheads at the cost of adding buffer management overheads, however if the buffer is not sufficient (i.e. buffer overflow), the sender would have to be blocked until some of the corresponding receive operations had been posted, thus freeing up buffer space, this can often lead to unforeseen overheads and performance degradation
pp.5, while buffering alleviates many of the deadlock situations, it is still possible to write code that deadlocks, this is due to the fact that as in the non-buffered case receive calls are always blocking

7.1.2.2 Non-blocking message passing operations

pp.5, non-blocking operations are generally accompanied by a check-status operation, which indicates whether it is safe for the programmer to touch this data, non-blocking operations can themselves be buffered or non-buffered; pp.6, with check-status the idling time when the process is waiting for the corresponding receive in a blocking operation can now be utilized for computation, provided it does not update the data being sent, blocking operations facilitate safe and easier programming and non-blocking operations are useful for performance optimization by masking communication overhead

7.1.3 MPI: the Message Passing Interface

pp.7, message-passing became the modern-age form of assembly language, in which every hardware vendor provided its own library, that performed very well on its own hardware, but was incompatible with the parallel computers offered by other vendors

7.1.3.1 Starting and terminating the MPI library

pp.8, both MPI_Init and MPI_Finalize must be called by all the processes. An MPI implementation is expected to remove from the argv array any command-line arguments that should be processed by the implementation before returning back to the program and to decrement argc accordingly

7.1.3.2 Communicators

pp.8, a communication domain is a set of processes that are allowed to communicate with each other, information about communication domains is stored in variables of type MPI_Comm that are called communicators; pp.8, MPI defines a default communicator called MPI_COMM_WORLD which includes all the processes involved in the parallel execution

7.1.3.3 Getting information

pp.8, MPI_Comm_size(MPI_Comm comm, int *size) returns in the variable size the number of processes that belong to the communicator comm, every process that belongs to a communicator is uniquely identified by its rank, and MPI_Comm_rank(MPI_Comm comm, int *rank) stores the rank of the process in the variable rank

7.1.3.4 Sending and receiving messages

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

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
MPI_status *status)

pp.9, the buffer consists of consecutive entries of the type specified by the parameter datatype, the number of entries in the buffer is given by the parameter count, forall C datatypes an equivalent MPI datatype is provided, however MPI allows two additional datatypes that are not part of the C language MPI_BYTE and MPI_PACKED, note that the length of the message in MPI_Send as well as in other MPI routines is specified in terms of the number of entries being sent and not in terms of the number of bytes, this improves portability since the number of bytes used to store various datatypes can be different for different architectures, each message has an integer-valued tag associated with it, this is used to distinguish different types of messages, if there are many messages ith identical tag from the same process, then any one of these messages is received

pp.10, MPI allows specification of wildcard arguments for both source and tag, if source is set to MPI_ANY_SOURCE, then any process of the communication domain can be the source of the message, similarly if tag is set to MPI_ANY_TAG, then messages with any tag are accepted, the count and datatype arguments of MPI_Recv are used to specify the length of the supplied buffer, the received message should be of length equal to or less than this length, if the received message is larger than the supplied buffer, then an overflow error will occur, in C status is stored using the MPI_status data-structure with three fields—MPI_SOURCE and MPI_TAG store the source and the tag of the received message, MPI_ERROR stores the error-code of the received message, the status argument also returns information about the length of the received message, this information is not directly accessible from the status variable, but it can be retrieved by calling the MPI_Get_count function

int MPI_Get_count(MPI_status *status, MPI_Datatype datatype, int *count)

pp.11, the MPI_Recv returns only after the requested message has been received and copied into the buffer, that is MPI_Recv is a blocking receive operation, on the other hand MPI allows two different implementations for MPI_Send

blocking: MPI_Send returns only after the corresponding MPI_Recv have been issued and the message has been sent to the receiver

non-blocking: MPI_Send first copies the message into a buffer and then returns, without waiting for the corresponding MPI_Recv to be executed

pp.12, deadlock can be corrected by matching the order in which the send and receive operations are issued, improper use of MPI_Send and MPI_Recv can also lead to deadlocks in situations when each processor needs to send and receive a message in a circular fashion, which can be avoided by partitioning the processes into two groups—odd-numbered processes sending followed by receiving and even-numbered processes receiving followed by sending; pp.12, this communication pattern appears frequently and for this reason MPI provides the MPI_Sendrecv function that both sends and receives a message

int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype senddatatype, int dest, int sendtag
void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, int recvtag,
MPI_Comm comm, MPI_Status *status)

where the source and destination of the messages can be the same or different, the requirement for the send and receive buffers of MPI_Sendrecv be disjoint can be alleviated by using MPI_Sendrecv_replace which performs a blocking send and receive with a single buffer

int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype senddatatype, int dest, int sendtag
int source, int recvtag, MPI_Comm comm, MPI_Status *status)

7.1.4 Topologies and Embedding

pp.16, it is up to the MPI library to find the most appropriate mapping that reduces the cost of sending and receiving messages

7.1.5 Creating and Using Cartesian Topologies

pp.16, MPI provides routines that allow the specification of virtual process topologies of arbitrary connectivity in terms of a graph, however most commonly used topologies in message-passing programs are Cartesian topologies, MPI's function for describing Cartesian topologies is called MPI_Cart_create

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
                    int reorder, MPI_Comm *comm_cart)
```

dims: specify the size along each dimension of the topology, the ith element of this array stores the size of the ith dimension of the topology, if the total number of processes specified in the dims array is smaller than the number of processes in the old communicator group, then some processes will not be part of the Cartesian topology, for this set of processes the value of comm_cart will be set to MPI_COMM_NULL (an MPI defined constant)

period: specify whether or not the topology has wraparound connections

reorder: determine if the processes in the new communicator group are to be reordered or not, if reorder is false then the rank of each process in the new communicator group is identical to its rank in the old group, otherwise MPI_Cart_create may reorder the processes if that leads to a better embedding of the virtual topology onto the parallel computer

pp.17, for performing coordinate-to-rank and rank-to-coordinate translations we have

```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank);
int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims, int *coords);
```

pp.17, MPI provides the following function to compute the rank of the source and destination for shifts along a dimension

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir, int s_step, int *rank_source, int *rank_dest)
```

7.1.6 Overlapping Communication with Computation

7.1.6.1 Non-blocking communication operations

pp.20, in order to overlap communication with computation MPI provides a pair of functions for performing non-blocking send and receive operations

MPI_Isend: starts a send operation but return before the data is copied out of the buffer

MPI_Irecv: starts a receive operation but returns before the data has been received and copied into the buffer

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm, MPI_request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_request *request)
```

pp.20, to check the completion of non-blocking send and receive operations, MPI provides a pair of functions MPI_Test and MPI_Wait, the main difference from blocking send and receive functions is

that they take an additional argument request, which is used as an argument in the MPI_Test and MPI_Wait functions, which returns the status information associated with the receive operation

int MPI_Test(MPI_request *request, int *flag, MPT_Status *status)
int MPI_Wait(MPI_request *request, MPI_Status *status)

in the case that the non-blocking operation has finished, the request object pointed to by request is deallocated and request is set to MPI_REQUEST_NULL, if the operation has not finished request is not modified and the value of the status object is undefined
pp.21, a non-blocking communication operation can be matched with a corresponding blocking operation (e.g. a non-blocking send + a blocking receive), by using non-blocking communication operations we can remove most of the deadlocks associated with their blocking counterparts (replace either the send or receive operations with their non-blocking counterparts)
pp.23, in order to overlap communication with computation we have to use auxiliary arrays, this is to ensure that incoming messages never overwrite the blocks that are used in the computation which proceeds concurrently with the data transfer, thus increased performance by overlapping communication with computation comes at the expense of increased memory requirements, this is a trade-off that is often made in message-passing programs

7.1.7 Collective Communication and Computation Operations

pp.24, even though collective communication operations do not act like barriers (a processor can go past its call for the collective communication operation before other processes have reached it), it acts like a virtual synchronization step in the following sense—the parallel program should be written such that it behaves correctly even if a global synchronization is performed before and after the collective call, since the operations are virtually synchronous, they do not require tags, for most collective communication operations, MPI provides two different variants, the first transfers equal-size data and the second transfers data that can be of different sizes

7.1.7.1 Barrier

pp.24, the barrier synchronization operation is performed in MPI using

int MPI_Barrier(MPI_Comm comm)

the call to MPI_Barrier returns only after all the processes in the group have called this function

7.1.7.2 Broadcast

pp.24, the one-to-all broadcast operation is performed in MPI using

int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)

the amount of data sent by the source process must be equal to the amount of data that is being received by each process, i.e. the count and datatype fileds must match on all processes

7.1.7.3 Reduction

pp.24, the all-to-one reduction operation is performed in MPI using

int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
MPI_Op op, int target, MPI_Comm comm)

it combines the elements stored in sendbuf of each process in the group using the operation specified in op and returns the combined values in recvbuf of the target process, all processes must provide a recvbuf array even if they are not the target of the reduction operation, all the processes must call MPI_Reduce with the same value for count, datatype, op, target, and comm, when count is more than one the combine operation is applied element-wise on each entry of the sequence
pp.25, one possible application of MPI_MAXLOC or MPI_MINLOC is to compute the maximum or minimum of a list of numbers each residing on a different process and also the rank of the first process that stores this maximum or minimum, since both MPI_MAXLOC and MPI_MINLOC require datatypes that correspond to pairs of values, a new set of MPI datatypes have been defined as shown in table 6.4, when the result of the reduction operation is needed by all the processes, MPI provides the MPI_Allreduce operation that returns the result to all the processes

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

7.1.7.4 Prefix

pp.26, the prefix-sum operation is performed in MPI using

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm)
```

the type of supported operations i.e. op as well as the restrictions on the various arguments of MPI_Scan are the same as those for the reduction operation MPI_Reduce

7.1.7.5 Gather

pp.27, the gather operation is performed in MPI using

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype senddatatype,
               void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
               int target, MPI_Comm comm)
```

the data from process with rank i are stored in the recvbuf starting at location $i*$sendcount, the data sent by each process must be of the same size and type, that is MPI_Gather must be called with the sendcount and senddatatype arguments having the same values at each process, the information about the receive buffer, its length and type applies only for the target process and is ignored for all the other processes, the argument recvcount specifies the number of elements received by each process and not the total number of elements it receives, so recvcount must be the same as sendcount and their datatypes must be matching
pp.27, MPI also provides the MPI_Allgather function in which the data are gathered to all the processes and not only at the target process

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype senddatatype,
                  void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
                  MPI_Comm comm)
```

pp.27, MPI also provides versions in which the size of the arrays can be different, MPI refers to these operations as the vector variants

```
int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype senddatatype,
                void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvdatatype,
                int target, MPI_Comm comm)
int MPI_Allgatherv(void *sendbuf, int sendcount, MPI_Datatype senddatatype,
                void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvdatatype,
                MPI_Comm comm)
```

the amount of data sent by process i is equal to recvcounts[i], the array parameter displs, which is of the same size as the communicator comm, is used to determine where in recvbuf the data sent by each process will be stored, in particular the data sent by process i are stored in recvbuf starting at location displs[i], as opposed to the non-vector variants the sendcount parameter can be different for different processes

7.1.7.6 Scatter

pp.28, the scatter operation is performed in MPI using

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype senddatatype,
                void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
                int source, MPI_Comm comm)
```

the data from process with rank i are stored in the recvbuf, process i receives sendcount contiguous elements of the type senddatatype starting from the $i*$sendcount location of the sendbuf of the source process, MPI_Scatter must be called by all the processes with the same values for the sendcount, senddatatype, recvcount, recvdatatype, source, and comm arguments,, note again that sendcount is the number of elements sent to each individual process

```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype senddatatype,
                void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
                int source, MPI_Comm comm)
```

the target process sends sendcounts[i] elements to process i, the array displs, which is of the same size as the communicator comm, is used to determine where in sendbuf these elements will be sent from, in particular the data sent to process i start at location displs[i] of array sendbuf, note that by appropriately setting the displs array we can use MPI_Scatterv to send overlapping regions of sendbuf

7.1.7.7 All-to-all

pp.28, the all-to-all personalized communication operation is performed in MPI by using

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype senddatatype,
                void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
                MPI_Comm comm)
```

each process sends to process i sendcount contiguous elements of type senddatatype starting from the i*sendcount location of its sendbuf array, each process receives from process i recvcount elements of type recvdatatype and stores them in its recvbuf array starting at location i*recvcount, MPI_Alltoall must be called by all the processes with the same values for the sendcount, senddatatype, recvcount, recvdatatype, and comm arguments
pp.28, MPI also provides a vector variant of the all-to-all personalized communication operation

```
int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls, MPI_Datatype senddatatype,
                  void *recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvdatatype,
                  MPI_Comm comm)
```

each process sends to process i, starting at location sdispls[i] of the array sendbuf, sendcounts[i] contiguous elements, each process receives from process i recvcounts[i] elements that are stored in contiguous locations of recvbuf starting at location rdispls[i], MPI_Alltoallv must be called by all the processes with the same values for the senddatatype, recvdatatype, and comm arguments
pp.31, comparing the row-wise and column-wise version of matrix-vector multiplication, a row-wise distribution is preferable as it leads to small communication overhead, however many times an application needs to compute not only Ax but also $A^T x$, it is much cheaper to use the program for the column-wise distribution than to transpose the matrix and then use the row-wise program, also note that using a dual of the all-gather operation, it is possible to develop a parallel formulation for column-wise distribution that is as fast as the program using row-wise distribution, however this dual operation is not available in MPI

7.1.8 Groups and Communicators

pp.35, MPI provides several mechanisms for partitioning the group of processes that belong to a communicator into subgroups, a general method for partitioning is

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

this function is a collective operation and thus needs to be called by all the processes in the communicator comm, each subgroup contains all processes that have supplied the same value for the color parameter, within each subgroup the processes are ranked in the order defined by the value of the key parameter, with ties broken according to their rank in the old communicator comm
pp.35, MPI provides the MPI_Cart_sub function that allows us to partition a Cartesian topology into sub-topologies that form lower-dimensional grids

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims, MPI_Comm *comm_subcart)
```

if keep_dims[i] is true then the ith dimension is retained in the new sub-topology
⇒ tips of boosting performance:

- use non-blocking send & recv to reduce idling overhead
- combine send & recv operation using sendrecv()
- overlap communication with computation through non-blocking send & recv and data buffer

8 Week 11 Articles

8.1 Introduction to Parallel Computing Chapter 2 Parallel Programming Platforms

8.1.1 Limitations of Memory System Performance

- pp.7, the fraction of data references satisfied by the cache is called the cache hit ratio of the computation, repeated reference to a data item in a small time window is called temporal locality of reference; pp.9, if the computation does not have spatial locality, then effective bandwidth can be much smaller than the peak bandwidth; pp.11, we define full-bandwidth as the rate of data transfer required by a computation to make it processor bound instead of memory bound

- pp.12, prefetching works for much the same reason as multithreading; pp.13, however it is important to realize that multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem

8.1.2 Dichotomy of Parallel Computing Platforms

pp.14, the logical organization refers to a programmer's view of the platform while the physical organization refers to the actual hardware organization of the platform, the two critical components of parallel computing from a programmer's perspective are (1) ways of expressing parallel tasks and (2) mechanisms for specifying interaction between these tasks, the former is sometimes also referred to as the control structure and the latter as the communication model

8.1.2.1 Communication model of parallel platforms

pp.17, there are two primary forms of data exchange between parallel tasks

shared-address-space platforms: pp.17, support a common data space that is accessible to all processors, if the time taken to access certain memory words is longer than others the platform is called a non-uniform memory access (NUMA) multicomputer (we define NUMA and UMA architectures only in terms of memory access times and not cache access times); pp.19, shared-memory computer in which each processor has equal access to any memory segment is identical to the UMA model, a distributed-memory shared-address-space computer is identical to a NUMA machine
pp.18, shared-address-space programming paradigms such as threads (POSIX, NT) and directives (OpenMP) support synchronization using locks and related mechanisms, supporting a shared-address-space in the presence of cache involves two major tasks: (1) providing an address translation (2) ensuring cache coherence

message-passing platforms: pp.19, since interactions are accomplished by sending and receiving messages, the basic operations in this programming paradigm are send and receive

8.1.3 Physical Organization of Parallel Platforms

8.1.3.1 Architecture of an ideal parallel computer

pp.20, all processors access the same address space, this ideal model is also referred to as a parallel random access machine (PRAM), PRAMs can be divided into four subclasses: EREW, CREW,

ERCW, CRCW, where R = read vs. W = write and E = exclusive vs. C = concurrent
pp.21, interconnection networks can be classified as

static/direct: consist of point-to-point communication links among processing nodes

dynamic/indirect: built using switches, communication links are connected to one another dynamically by the switches to establish paths

8.1.3.2 Network topologies

bus-based network: pp.23, a shared medium that is common to all the nodes

crossbar network: pp.24, employs a grid of switches to connect p processors to b memory banks

multistage network: pp.25, the crossbar interconnection network is scalable in terms of performance but unscalable in terms of cost, conversely the shared bus network is scalable in terms of cost but unscalable in terms of performance, an intermediate class of networks called multistage interconnection networks lies between these two extremes

completely-connected network: pp.28, each node has a direct communication link to every other node in the network

star-connected network: pp.28, one processor acts as the central processor, every other processor has a communication link connecting it to this processor

linear array and mesh: pp.29, due to the large number of links in completely connected networks, sparser networks are typically used to build parallel computers
pp.29, a variety of physical simulations commonly executed on parallel computers (e.g. 3D weather modeling) can be mapped naturally to 3D network topologies, for this reason 3D cubes are used commonly in interconnection networks for parallel computers
pp.30, the hypercube topology has two nodes along each dimension and $\log p$ dimensions, in general a d-dimensional hypercube is constructed by connecting corresponding nodes of two $(d-1)$ dimensional hypercubes

tree-based network: pp.30, there is only one path between any pair of nodes, both linear arrays and star-connected networks are special cases of tree networks; pp.31, tree networks suffer from a communication bottleneck at higher levels of the tree, this problem can be alleviated in dynamic tree networks by increasing the number of communication links and switching nodes closer to the root

8.1.3.3 Evaluating static interconnection networks

diameter: pp.32, the diameter is the maximum distance between any two processing nodes in the network, the distance between two processing nodes is defined as the shortest path between them

connectivity: pp.32, measure of the multiplicity of paths between any two processing nodes, a network with high connectivity is desirable because it lowers contention for communication resources, one measure of connectivity is the minimum number of arcs that must be removed from the network to break it into two disconnected networks

bisection width: pp.32, the minimum number of communication links that must be removed to partition the network into two equal halves; pp.33, bisection bandwidth is the product of the bisection width and the channel bandwidth

channel bandwidth: pp.32, channel width is equal to the number of physical wires in each communication link, the peak rate at which a single physical wire can deliver bits is called the channel rate, the peak rate at which data can be communicated between the ends of a communication link is called channel bandwidth, channel bandwidth is the produce of channel rate and channel width

8.1.3.4 Cache coherence in multiprocessor systems

pp.35, when a processor changes the value of its copy of the variable, one of two things must happen: (1) the other copies must be invalidated (2) the other copies must be updated; pp.36, the tradeoff between invalidate and update schemes is the classic tradeoff between communication overhead (updates) and idling (stalling in invalidates)
pp.36, false sharing refers to the situation in which different processors update different parts of the same cache-line

8.1.4 Communication Costs in Parallel Machines

8.1.5 Routing Mechanisms for Interconnecton Networks

8.1.6 Impact of Process-Processor Mapping and Mapping Techniques

pp.61, it is possible to map denser networks into sparser networks with associated congestion overheads, this implies that a sparser network whose link bandwidth is increased to compensate for the congestion can be expected to perform as well as the denser network

8.2 Introduction to Parallel Computing Chapter 8 Dense Matrix Algorithms

8.2.1 Matrix-Vector Multiplication

8.2.1.1 Rowwise 1D partitioning

pp.2, the parallel algorithm is cost-optimal because the complexity of the serial algorithm is $\Theta(n^2)$, the asymptotic isoefficiency function of the parallel matrix-vector multiplication algorithm with 1D partitioning is $\Theta(p^2)$

8.2.1.2 2D partitioning

pp.3, the algorithm is not cost-optimal because the cost is $\Theta(n^2 \log n)$; pp.4, the overall asymptotic isoefficiency function is given by $\Theta(p \log^2 p)$; pp.5, among the two partitioning schemes 2D partitioning has a better (smaller) asymptotic isoefficiency function

8.2.2 Matrix-Matrix Multiplication

8.2.2.1 Cannon's algorithm

pp.6, after a submatrix multiplication step, each block of A moves one step left and each block of B moves one step up ($\Rightarrow$ instead of letting the processors moving along the blocks, let the blocks move into the processors)

8.2.2.2 The DNS algorithm

pp.8, the parallel run time for multiplying two $n \times n$ matrices using the DNS algorithm on n^3 processes is $\Theta(\log n)$, the DNS algorithm is not cost-optimal for n^3 processes since its process-time product of $\Theta(n^3 \log n)$ exceeds the $O(n^3)$ sequential complexity of matrix multiplication
pp.8, the isoefficiency function of the block version of the DNS algorithm is $\Theta(p \log^3 p)$, the algorithm is cost-optimal for $n^3 = \Omega(p \log^3 p)$

8.2.3 Solving a System of Linear Equations

8.2.3.1 A simple Gaussian elimination algorithm

pp.9, Gaussian elimination involves approximately $n^2/2$ divisions and approximately $(n^3/3) - (n^2/2)$ subtractions and multiplications, with the assumption that each scalar arithmetic operation takes unit time, the sequential run time of the procedure is approximately $2n^3/3$ (for large n)

9 Week 12 Article

9.1 Introduction to Parallel Computing Chapter 9 Sorting

pp.1, sorting algorithms are categorized as internal or external

internal: the number of elements to be sorted is small enough to fit into the process's main memory

external: use auxiliary storage for sorting because the number of elements to be sorted is too large to fit into memory

this chapter concentrates on internal sorting algorithms only
pp.1, sorting algorithms can be categorized as comparison-based and noncomparison-based

comparison-based: repeatedly comparing pairs of elements and if they are out of order exchanging them (called compare-exchange), the lower bound on the sequential complexity of any sorting algorithms that is comparison-based is $\Theta(n \log n)$

noncomparison-based: using certain known properties of the elements such as their binary representation or their distribution, the lower-bound complexity of these algorithms is $\Theta(n)$

we concentrate on comparison-based sorting algorithms in this chapter

9.1.1 Issues in Sorting on Parallel Computers

9.1.1.1 Where the input and output sequences are stored

pp.2, in this chapter we assume that the input and sorted sequences are distributed among the processes

9.1.1.2 How comparisons are performed

one element per process: pp.2, consider the case in which each process holds only one element of the sequence to be sorted, we can perform comparison by having both processes send their elements to each other, in today's parallel computers it takes more time to send an element from one process to another than it takes to compare the elements, consequently any parallel sorting formation that uses as many processes as elements to be sorted will deliver very poor performance because the overall parallel run time will be dominated by interprocess communication

more than one element per process: pp.3, as in the one-element-per-process case, two processes P_i and P_j may have to redistribute their blocks of n/p elements, if the block of n/p elements at each process is already sorted, the redistribution can be done efficiently as follows—each process sends its block to the other process, each process merges the two sorted blocks and retains only the appropriate half of the merged block—compare-split (vs. compare-exchange)

pp.3, as the block size increases, the significance of the message startup time t_s decreases and for sufficiently large blocks it can be ignored, thus the time required to merge two sorted blocks of n/p elements is $\Theta(n/p)$

9.1.2 Sorting Networks

9.1.2.1 Bitonic sort

pp.9, the last stage of an n-element bitonic sorting network contains a bitonic merging network with n inputs, this has a depth of $\log n$, the other stages perform a complete sort of $n/2$ elements, hence the depth $d(n)$ of the network is given by the following recurrence relation $d(n) = d(n/2) + \log n$, solving it we obtain

$$d(n) = \sum_{i=1}^{\log n} i = \frac{1}{2}\left(\log^2 n + \log n\right) = \Theta(\log^2 n)$$

this network can be implemented on a serial computer yielding a $\Theta(n \log^2 n)$ sorting algorithm

9.1.2.2 Mapping bitonic sort to hypercube and a mesh

pp.9, one of the key aspects of the bitonic algorithm is that it is communication intensive

one element per process: pp.9, ideally wires that perform a compare-exchange should be mapped onto neighboring processes, in any step the compare-exchange operation is performed between two wires only if their labels differ in exactly one bit, wires whose labels differ in the least-significant bit perform a compare-exchange in the last step of each stage, in general wires whose labels differ in the ith least significant bit perform a compare-exchange $(\log n - i + 1)$ times

hypercube: pp.10, in a hypercube processes whose labels differ in only one bit are neighbors; pp.11, a bitonic merge of sequences of size 2^k can be performed on a k-dimensional subcube, furthermore during the ith step fo this bitonic merge the processes that compare their elements are neighbors along the $(k - (i - 1))$th dimension

pp.11, the pseudocode of parallel formulation of bitonic sort on a hypercube with $n = 2^d$ processes is (where d is the dimension of the hypercube)

```
procedure bitonic_sort(p, d)
for i ← 1 to d − 1 do
    for j ← i down to 0 do
        if (i + 1)th bit of pth process ≠ jth bit of pth process
            compare_exchange max(j)
        else
            compare_exchange min(j)
```

pp.12, during each step of the algorithm every process performs a compare-exchange operation, the algorithm performs a total of $(1 + \log n) \log n/2$ such steps, thus the parallel run time is $T_P = \Theta(\log^2 n)$, this parallel formulation of bitonic sort is cost optimal with respect to the sequential implementation of bitonic sort $\Theta(n \log^2 n)$, but it is not cost-optimal with respect to an optimal comparison-based sorting algorithm which has a serial time complexity of $\Theta(n \log n)$

mesh: pp.12, the connectivity of a mesh is lower than that of a hypercube, so it is impossible to map wires to processes such that each compare-exchange operation occurs only between neighboring processes, instead we map wires such that the most frequent compare-exchange operations occur between neighboring processes, in this chapter we concentrate on the row-major shuffled mapping, the advantage of row-major shuffled mapping is that processes that perform compare-exchange operations reside on square subsections of the mesh whose size is inversely related to the frequency of compare-exchange, in general wires that differ in the ith least significant bit are mapped onto mesh processes that are $2^{\lfloor (i-1)/2 \rfloor}$ communication links away, during the $(1 + \log n) \log n/2$ steps of the algorithm the total amount of communication performed by each process is

$$T_{\text{comm}} = \sum_{i=1}^{\log n} \sum_{j=1}^{i} 2^{\lfloor (j-1)/2 \rfloor} \approx 7\sqrt{n} = \Theta(\sqrt{n})$$

during each step of the algorithm each process performs at most one comparison, thus the total computation performed by each process is

$$T_{\text{comp}} = \Theta(\log^2 n)$$

this yields a parallel run time of

$$T_P = \Theta(\log^2 n) + \Theta(\sqrt{n})$$

this is not a cost-optimal formulation, because the process-time product is $\Theta(n^{1.5})$ but the sequential complexity of sorting is $\Theta(n \log n)$, although the parallel formulation for a hypercube was optimal with respect to the sequential complexity of bitonic sort, the formulation for mesh is not, and we cannot do any better because the element stored in the process at the upper-left corner must travel along $2\sqrt{n} - 1$ communication links to end up in the process at the lower-right corner, thus the run time of sorting on a mesh is bounded by $\Omega(\sqrt{n})$, and our parallel formulation achieves this lower bound

a block of elements per process: pp.13, one way to obtain a parallel formulation with our new setup is to think of each process as consisting of n/p smaller processes, in other words imagine emulating n/p processes by using a single process, this virtual process approach leads to a poor parallel implementation of bitonic sort, to see this consider the case of a hypercube with p processes, its run time will be $\Theta((n \log^2 n)/p)$, which is not cost-optimal because the process-time product is $\Theta(n \log^2 n)$

pp.14, an alternative way of dealing with blocks of elements is to use the compare-split operation, since the total number of blocks is p, the bitonic sort algorithm has a total of $(1 + \log p) \log p/2$ steps, the main difference between this formulation and the one that uses virtual processes is that the n/p elements assigned to each process are initially sorted locally using a fast sequential sorting algorithm, this initial local sort makes the new formulation more efficient and cost-optimal

hypercube: the compare-exchange operations are replaced by compare-split operations, each taking $\Theta(n/p)$ computation time and $\Theta(n/p)$ communication time, the parallel run time of this formulation is

$$T_P = \underbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}_{\text{local sort}} + \underbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}_{\text{comparison}} + \underbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}_{\text{communication}}$$

mesh: similar to the one-element-per-process case

$$T_P = \underbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}_{\text{local sort}} + \underbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}_{\text{comparison}} + \underbrace{\Theta\left(\frac{n}{\sqrt{p}}\right)}_{\text{communication}}$$

pp.15, from the analysis for hypercube and mesh we see that parallel formulations of bitonic sort are neither very efficient nor very scalable, this is primarily because the sequential algorithm is suboptimal

9.1.3 Bubble Sort and its Variants

pp.15, since serial algorithms with $\Theta(n \log n)$ time complexity exist, we should be able to use $\Theta(n)$ processes to sort n elements in time $\Theta(\log n)$

9.1.3.1 Odd-even transposition

pp.16, during the odd phase elements with odd indices are compared with their right neighbors, similarly during the even phase elements with even indices are compared with their right neighbors, each phase of the algorithm (either odd or even) requires $\Theta(n)$ comparisons and there are a total of n phases, thus the sequential complexity is $\Theta(n^2)$

parallel formulation: pp.18, it is easy to parallelize odd-even transposition sort, during the odd phase each process that has an odd label compare-exchanges its element with the element residing on its right neighbor, similarly during the even phase each process with an even label compare-exchanges its element with the element of its right neighbor, this requires time $\Theta(1)$, this formulation of odd-even transposition sort is not cost-optimal because its process-time product is $\Theta(n^2)$

pp.18, to obtain a cost-optimal parallel formulation we use fewer processes, each process is assigned a block of n/p elements, which it sorts internally using merge sort or quicksort in $\Theta((n/p)\log(n/p))$ time

shellsort: pp.19, the main limitation of odd-even transposition sort is that it moves elements only one position at a time, we need an algorithm that moves elements long distances, shellsort is one such serial sorting algorithm, the algorithm consists of two phases

1. processes that are far away from each other in the array compare-split their elements, elements thus move long distances to get close to their final destinations in a few steps
2. the algorithm switches to an odd-even transposition sort similar to the one described in the previous section, the only difference is that the odd and even phases are performed only as long as the blocks on the processes are changing

9.1.4 Quicksort

pp.20, the quicksort algorithm has an average complexity of $\Theta(n\log n)$, the lower bound for comparison-based sorting, it is a divide-and-conquer algorithm that sorts a sequence by recursively dividing it into smaller subsequences; pp.22, the complexity of partitioning a sequence of size k is $\Theta(k)$, quicksort's performance is greatly affected by the way it partitions a sequence, splitting into two roughly equal-size subsequences yields the optimal $\Theta(n\log n)$, although quicksort can have $O(n^2)$ worst-case complexity, the average number of compare-exchange operations needed by quicksort for sorting a randomly-ordered input sequence is $1.4n\log n$ which is asymptotically optimal

9.1.4.1 Parallelize quicksort

pp.22, one way to parallelize quicksort is to execute it initially on a single process, then hen the algorithm performs its recursive calls, assign one of the subproblems to another process, its major drawback is that partitioning the array into two smaller arrays is done by a single process i.e. it performs the partitioning step serially, the run time of this formulation is thus bounded below by $\Omega(n)$, this formulation is not cost-optimal because its process-time product is $\Omega(n^2)$
pp.22, performing partitioning in parallel is essential in obtaining an efficient parallel quicksort, because in the recurrence equation $T(n) = 2T(n/2) + \Theta(n)$, the term $\Theta(n)$ is due to the partitioning of the array, if the partitioning step is performed in time $\Theta(1)$ using $\Theta(n)$ processes, it is possible to obtain an overall parallel run time of $\Theta(\log n)$ which leads to a cost-optimal formulation, however without parallelizing the partitioning step the best we can do while maintaining cost-optimality is to use only $\Theta(\log n)$ processes to sort n elements in time $\Theta(n)$ (ref. problem 9.14); pp.23, however partitioning in time $\Theta(1)$ by using $\Theta(n)$ processes is difficult for most parallel computing models, the only known algorithms are for the abstract PRAM models, because of communication overhead the partitioning step takes longer than $\Theta(1)$ on realistic shared-address-space and message-passing parallel computers

9.1.4.2 Parallel formulation for a CRCW PRAM

pp.23, executing quicksort can be visualized as constructing a binary tree, in this tree the pivot is the root, elements smaller than or equal to the pivot go to the left subtree, and elements larger than the pivot go to the right subtree, subsequent pivot elements, one for each new subarray, are then selected in parallel; pp.24, the algorithm continues until n pivot elements are selected,

a process exits when its element becomes a pivot, during each iteration of the algorithm a level of the tree is constructed in time $\Theta(1)$, thus the average complexity of the binary tree building algorithm is $\Theta(\log n)$; pp.27, after building the binary tree the algorithm traverses the tree and keeps a count of the number of elements in the left and right subtrees of any element

9.1.4.3 Parallel formulation for practical architectures

shared-address-space parallel formulation: pp.27, the partitioning of processes into two groups is done according to the relative sizes of the S and L blocks, in particular the first $\lceil |S|p/n + 0.5 \rceil$ processes are assigned to sort the smaller elements S, the recursion ends when a particular sub-block of elements is assigned to only a single process i.e. when the array is split into p parts (ref. pp.30), in which case the process sorts the elements using a serial quicksort algorithm
pp.26, the labels of the processes define the global order of the sorted sequence; pp.28, S is obtained by concatenating the various S_i blocks over all the processes in increasing order of process label, similarly L is obtained by concatenating the various L_i blocks in the same order, as a result for process P_i the jth element of its S_i sub-block will be stored at location $\sum_{k=0}^{j-1} |S_k| + j$ and the jth element of its L_i sub-block will be stored at location $\sum_{k=i}^{p-1} |L_k| - j$, these locations can be easily computed using the prefix-sum operation
pp.30, the communication overhead in the above formulation is reflected in the $\Theta(\log^2 p)$ term, it is interesting to note that the overall scalability of the algorithm is determined by the amount of time required to perform the pivot broadcast (which takes $\Theta(\log p)$ time using an efficient recursive doubling approach) and the prefix sum operations (since we need two prefix sum, one for S one for L, each of which takes $\Theta(\log p)$ time)

message-passing parallel formulation: pp.30, similar to the shared-address-space case, the array is also partitioned around a particular pivot element using a two-phase approach

1. similar to the shared-address-space formulation, partition into the S_i and L_i sub-arrays locally
2. determine which processes will be responsible for $S = \cup_i S_i$ and $L = \cup_i L_i$, and the processes send their S_i and L_i arrays to the corresponding processes, the method used to determine which processes will be responsible for sorting S and L is identical to that for the shared-address-space formulation

pp.31, since in general the communication step involves all-to-all personalized communication (because a particular process may end up receiving elements from all other processes), the amount of time required for sending and receiving the various arrays has a lower bound of $\Theta(n/p)$, thus the overall complexity for the split is $\Theta(n/p) + \Theta(\log p)$ which is asymptotically similar to that of the shared-address-space formulation

9.1.4.4 Pivot selection

pp.32, one way to select pivots is that during the ith split one process in each of the process groups randomly selects one of its elements to be the pivot for this partition, this is analogous to the random pivot selection in the sequential quicksort algorithm, however it is not well suited to the parallel formulation, because one poor pivot may lead to a partitioning in which a process becomes idle, and that will persist throughout the execution of the algorithm
pp.32, if the initial distribution of elements in each processes is uniform, the median of each

n/p-element subsequence is very close to the median of the entire n-element sequence, since the distribution of elements on each process is the same as the overall distribution of the n elements, and the split not only maintains load balance but also preserves the assumption of uniform element distribution in the process group

9.1.5 Bucket and Sample Sort

bucket sort: pp.32, a popular serial algorithm for sorting an array of n elements whose values are uniformly distributed over an interval $[a, b]$ is the bucket sort algorithm, in this algorithm the interval $[a, b]$ is divided into m equal-sized subintervals referred to as buckets, the run time of this algorithm is $\Theta(n \log(n/m))$, for $m = \Theta(n)$ it exhibits linear run time $\Theta(n)$
pp.33, the parallel formulation of bucket sort consists of three steps

1. each process partitions its block of n/p elements into p sub-blocks
2. each process sends sub-blocks to the appropriate processes
3. each process sorts its bucket internally by using an optimal sequential sorting algorithm

sample sort: pp.33, unfortunately in most cases the actual input may not have a uniform distribution or its distribution may be unknown, in such situations an algorithm called sample sort ill yield significantly better performance ($\Rightarrow$ idea: from population distribution to sample distribution), a sample of size s is selected from the n-element sequence, and the range of the buckets is determined by sorting the sample and choosing $m - 1$ elements from the result, these elements (called splitters) divide the sample into m equal-sized buckets ($\Rightarrow$ from equal width to equal size), after defining the buckets the algorithm proceeds in the same way as bucket sorts, the performance of sample sort depends on (1) the sample size s and (2) the way it is selected from the n-element sequence
pp.33, the following splitter selection scheme guarantees that the number of elements end up in each bucket is roughly the same for all buckets, it divides the n elements into m blocks of size n/m each, and sorts each block by using quicksort, from each sorted block it chooses $m - 1$ evenly spaced elements, the $m(m - 1)$ elements selected from all the blocks represent the sample used to determine the buckets, this scheme guarantees that the number of elements ending up in each bucket is less than $2n/m$, this splitter selection scheme can be parallelized as follows

1. each process is assigned a block of n/p elements which it sorts sequentially
2. each process chooses $p - 1$ evenly spaced elements from the sorted block and sends them to one process say P_0
3. P_0 sequentially sorts the $p(p - 1)$ sample elements
4. P_0 selects the $p - 1$ splitters, and broadcasts them to all the other processes
5. proceeds in a manner identical to that of bucket sort

pp.34, the complexity of each step is as follows

1. internal sort of n/p elements requires time $\Theta((n/p) \log(n/p))$
2. the selection of $p - 1$ samples requires time $\Theta(p)$, sending $p - 1$ elements to process P_0 is similar to a gather operation which requires time $\Theta(p^2)$

3. internal sort of $p(p-1)$ samples at P_0 requires time $\Theta(p^2 \log p)$
4. the selection of $p-1$ splitters requires time $\Theta(p)$, sending $p-1$ splitters to all the other processes via one-to-all broadcast requires time $\Theta(p \log p)$
5. each process can insert these $p-1$ splitters in its local sorted block of size n/p by performing $p-1$ binary searches, each process can then partition its block into p sub-blocks which requires time $\Theta(p \log(n/p))$, each process then sends sub-blocks to the appropriate processes/buckets the communication time of which depends on the size of the sub-blocks to be communicated and has an upper bound $O(n) + O(p \log p)$

pp.34, if we assume that the elements stored in each process are uniformly distributed, then each sub-block has roughly $\Theta(n/p^2)$ elements, in this case the parallel run time is

$$T_P = \underbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}_{\text{local sort}} + \underbrace{\Theta\left(p^2 \log p\right)}_{\text{sort sample}} + \underbrace{\Theta\left(p \log \frac{n}{p}\right)}_{\text{block partition}} + \underbrace{\Theta(n/p) + \Theta\left(p \log p\right)}_{\text{communication}}$$

9.1.6 Other Sorting Algorithms

9.1.6.1 Enumeration sort

pp.35, the basic idea behind enumeration sort is to determine the rank of each element, the rank of an element a_i is the number of elements smaller than a_i in the sequence to be sorted, the rank of a_i can be used to place it in its correct position in the sorted sequence

9.1.6.2 Radix sort

pp.36, the radix sort algorithm relies on the binary representation of the elements to be sorted

10 Week 13 Article

10.1 Parallel Breadth-First Search on Distributed Memory Systems

abstract: pp.1 LHS, we present two highly-tuned parallel approaches for BFS on large parallel systems: (1) a level-synchronous strategy that relies on a simple vertex-based partitioning of the graph (2) a two-dimensional sparse matrix partitioning-based approach that mitigates parallel communication overhead

introduction: pp.1 RHS, graph traversal problems such as BFS are by definition predominantly memory access-bound, and these accesses are further dependent on the structure of the input graph, thereby making the algorithm "irregular", we utilize a testbed of large-scale graphs which are all sparse i.e.

- the number of edges m is just a constant factor times the number of vertices n
- the average path length is a small constant value compared to the number of vertices, or is at most bounded by $\log n$

breadth-first search overview: pp.3 LHS, in multicore systems due to the memory-intensive nature of BFS, performance is still quite dependent on the graph size as well as the sizes

and memory bandwidths of the various levels of the cache hierarchy; pp.3 RHS, interprocessor communication is considered a significant performance bottleneck in prior work on distributed graph algorithms, the authors observe that a two-dimensional graph partitioning scheme would limit key collective communication phases of the algorithms to at most $\sqrt{p}$ processors, thus avoiding the expensive all-to-all communication steps
pp.4 LHS, a sparse graph can analogously be viewed as a sparse matrix, and optimization strategies for linear algebra computations similar to BFS, such as sparse matrix-vector multiplication, may be translated to the realm of graph algorithms to improve BFS performance as well

breadth-first search on distributed memory systems: pp.4 RHS, a natural way of distributing the vertices and edges of a graph on a distributed memory system is to let each processor own n/p vertices and all the outgoing edges from those vertices, we refer to this partitioning of the graph as 1D partitioning, as it translates to the one-dimensional decomposition of the adjacency matrix corresponding to the graph
pp.4 RHS, our sparse matrix approach uses the simple checkerboard partitioning; pp.5 RHS, there are two distinct communication phases in a BFS algorithm with 2D partitioning

expand: a pre-computation "expand" phase over the processor column using MPI_Allgather

fold: a post-computation "fold" phase over the processor row using MPI_Alltoallv

implementation details: pp.6 LHS, we use a stack in the 1D implementation and a sorted sparse vector in the 2D implementation, any extra data that are piggybacked to the frontier vectors adversely affect the performance, since the communication volume of the BFS benchmark is directly proportional to the size of this vector
pp.7 LHS, we achieve a reasonable load-balanced graph traversal by randomly shuffling all the vertex identifiers prior to partitioning, the downside to this is that the edge cut can be potentially as high as an average random balanced cut

algorithm analysis: pp.7 LHS, the PRAM asymptotic time complexity for a level-synchronous parallel BFS is $O(D)$ where D is the diameter of the graph, and the work complexity is $O(m + n)$
pp.7 RHS, the higher number of cache misses associated with larger working sets is perhaps the primary reason for the relatively higher computation costs of the 2D algorithm, most of the costs due to remote memory accesses is concentrated in two operations (1) the expand phase characterized by an Allgatherv operation over the processor column and (2) the fold phase characterized by an Alltoallv operation over the processor row, for large p the expand phase is likely to be more expensive than the fold phase

experimental studies: pp.8 LHS, we use undirected graphs for all our experiments, but the BFS approaches can work with directed graphs as well
pp.9 RHS, the performance margin between the 1D algorithm and the 2D algorithm increases in favor of the 1D algorithm as the graph gets sparser, the empirical data supports our analysis which concluded that the 2D algorithm performance was limited by the local memory accesses to its relatively larger vectors

conclusions and future work: pp.10 RHS, if the graph is undirected then one can save 50% space by storing only the upper/lower triangle of the sparse adjacency matrix, effectively

doubling the size of the maximum problem that can be solved in-memory on a particular system
pp.10 RHS, an alternative to randomization of vertex identifiers is to use hypergraph partitioning software to reduce communication

11 Week 14 Articles

11.1 A Separator Theorem for Planar Graphs

abstract: pp.1, the vertices of G can be partitioned into three sets A, B, C such that

- no edge joins a vertex in A with a vertex in B
- neither A nor B contains more than $2n/3$ vertices
- C contains no more than $2\sqrt{2}\sqrt{n}$ vertices

we exhibit an algorithm which finds such a partition A, B, C in $O(n)$ time

separator theorems: pp.2, to prove our results we need to use three facts about planarity, theorem 1 Jordan curve theorem

algorithm: pp.8, the proof of theorem 4 leads to an $O(n)$-time algorithm for finding a vertex partition satisfying the theorem, which requires a good representation of a planar embedding of a graph and a breadth-first spanning tree

11.2 An Efficient Heuristic Procedure for Partitioning Graphs

motivation: pp.291, this problem arises in several physical situations—for example in assigning the components of electronic circuits to circuit boards to minimize the number of connections between boards

introduction: pp.292, minimizing external cost is equivalent to maximizing internal cost because the total cost of all edges is constant
pp.292, because it seems likely that any direct approach to finding an optimal solution will require an inordinate amount of computation, we turn to an examination of heuristics

two-way uniform partitions: pp.295, the solution provides the basis for solving more general partitioning problems, in essence the method is this: starting with any arbitrary partition A, B of S, try to decrease the initial external cost T by a series of interchanges of subsets of A and B, when no further improvement is possible the resulting partition A', B' is locally minimum with respect to the algorithm, we shall indicate that the resulting partition has a fairly high probability of being a globally minimum partition, this process can then be repeated with the generation of another arbitrary starting partition A, B, and so on to obtain as many locally minimum partitions as we desire
pp.298, the process does not terminate immediately when some g_i is negative, this means that the process can sequentially identify sets for which the exchange of only a few elements would actually increase the cost, while the interchange of the entire sets produces a net gain
pp.298, numerous experiments have been performed to evaluate the procedure on different types of cost matrices

pp.299, a useful measure of the power of a heuristic procedure is the probability that if finds an optimal solution in a single trial
pp.299, define a pass to be the operations involved in making one cycle of identification of (a'_1, b'_1), ..., (a'_n, b'_n) and selection of sets X and Y to be exchanged, the total updating time in one pass grows as $(n-1)+(n-2)+\cdots+2+1 = O(n^2)$; pp.300, sorting is an $n \log n$ operation so in this method the total time required to sort D values in a pass will be approximately $n \log n + (n-1)\log(n-1) + \cdots + 2 \log 2 = O(n^2 \log n)$

multiple-way partitions: pp.303, the essential idea is to start with some partition into k sets of size n and by repeated application of the 2-way partitioning procedure to pairs of subsets, make the partition as close as possible to being pairwise optimal, of course pairwise optimality is only a necessary condition for global optimality

11.3 Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs

abstract: pp.1, the multilevel k-way partitioning algorithm reduces the size of the graph by collapsing vertices and edges (coarsening phase), finds a k-way partition of the smaller graph, and then it constructs a k-way partition for the original graph by projecting and refining the partition to successively finer graphs (uncoarsening phase), a key innovative feature of our parallel formulation is that it utilizes graph coloring to effectively parallelize both the coarsening and the refinement during the uncoarsening phase

multilevel k-way graph partitioning: pp.3, a multilevel k-way partition algorithm works as follows

coarsening phase: maximal matchings can be computed in different ways, the matching scheme that we use is called heavy-edge matching (HEM)

partitioning phase: in our partition algorithm, the k-way partition of G_m is computed using our multilevel recursive bisection algorithm

uncoarsening phase: the projection of the partition from G_{i+1} to G_i is constructed by simply assigning the vertices in U to the same part in G_i to the same part that vertex u belongs in G_{i+1}
even though the partition of G_{i+1} is at a local minima, the projected partition of G_i may not, hence it may still be possible to improve the projected partition by local refinement heuristics, the multilevel k-way partitioning algorithm uses a variation of the Kernighan-Lin algorithm extended to provide k-way partition refinement, this algorithm called greedy refinement (GR) is based on a simplified version of the Kernighan-Lin algorithm

parallel formulation: pp.6, out of the three phases of the multilevel k-way partitioning algorithm described above, the coarsening and the uncoarsening phases require the bulk of the computation (over 95%)
pp.7, in each iteration a maximal independent set of vertices I is selected using a variation of Luby's algorithm, all vertices in this independent set are assigned the same color

coarsening phase: our parallel matching algorithm is based on an extension of the serial algorithm that utilizes graph coloring to structure the sequence of computations

partitioning phase: we also parallelize this phase by using a parallel algorithm that parallelizes the recursive nature of the algorithm, each processor explores only a single path of the recursive bisection tree

uncoarsening phase: the single phase of the refinement algorithm is broken up into c sub-phases where c is the number of colors of the graph to be refined

conclusion: pp.18, the concurrency that is exposed by using coloring can also be used to implement more sophisticated algorithms

11.4 Algebraic Connectivity of Graphs

introduction: pp.298, let $n \geq 2$ and $0 = \lambda_1 \leq \lambda_2 \leq \lambda_3 \leq \cdots \leq \lambda_n$ be the eigenvalues of the matrix $A(G)$, from the Perron-Frobenius theorem applied to the matrix $(n-1)I - A(G)$ it follows that $\lambda_2 = 0$ if and only if the graph G is not connected

11.5 Partitioning Sparse Matrices with Eigenvectors of Graphs

abstract: pp.430, it is shown that lower bounds on separator sizes can be obtained in terms of the eigenvalues of the Laplacian matrix associated with a graph, a heuristic algorithm is designed to compute a vertex separator in a general graph by first computing an edge separator in the graph from an eigenvector of the Laplacian matrix, and then using a maximum matching in a subgraph to compute the vertex separator

introduction: pp.431, the spectral algorithm for computing vertex separators considered in this paper has three features that distinguish it from previous algorithms

- previous algorithms for computing separators such as the level-structure separator algorithm in Sparspak or the Kernighan-Lin algorithm make use of local information in the graph vs. the spectral method employs global information about the graph
- we can view the spectral method as an approach in which a vertex in the graph makes a continuous choice about which part in the initial partition it is going to belong to since all vertices with weights below the median weight form one part and the rest the other part vs. in the Kernighan-Lin method each vertex makes a discrete choice (zero or one) to belong to one set
- the dominant computation in the spectral method is an eigenvector computation by a Lanczos or similar algorithm

background: pp.432, the eigenvectors of the adjacency matrix corresponding to its algebraically largest eigenvalues have also been used to partition graphs, it is of interest to ask if a similar theorem holds for an eigenvector corresponding to the second largest eigenvalue of the adjacency matrix

a spectral partitioning algorithm: pp.441, the median component of the eigenvector can be obtained by an algorithm that selects the kth element out of n, this can be done in $O(n)$ tie in the worst case by a well-known algorithm of Blum, Floyd, Pratt, Rivest, and Tarjan, this algorithm finds the desired element by repeatedly partitioning a subarray with respect to a pivot element without sorting the array

pp.442, we used a less sophisticated median-finding algorithm which is $O(n)$ in the average-case and $O(n^2)$ in the worst-case, in practice the dominant step in the spectral partitioning algorithms is the computation of a second eigenvector by the Lanzcos algorithm

results: pp.444, for most problems the spectral algorithm also finds smaller edge separators in the graph than the Sparspak level-structure separator algorithm, there are a few problems where the best edge separator obtained by the latter algorithm is smaller than that obtained by the spectral algorithm, but the former edge separators separate the graph into parts with widely differing sizes
pp.446, Gilber and Zmijewski have observed that the quality of the partition found by the Kernighan-Lin algorithm strongly depends on the quality of the initial partition, on these four problems the Kernighan-Lin algorithm ran on the average about 3.2 times faster when the spectral partition was used, thus the spectral algorithm could be used to generate initial partitions of high quality for the Kernighan-Lin algorithm

convergence: pp.447, the dominant computation in the spectral partitioning algorithm is the computation of the second eigenvector of the Laplacian matrix by the Lanzcos algorithm, since the Lanzcos algorithm is an iterative algorithm, the number of iterations and the time required to compute this eigenvector is dependent on the number of correct digits needed in the eigenvector components

11.6 An Experimental Comparison of Pregel-like Graph Processing Systems

abstract: pp.1 LHS, the introduction of Google's Pregel generated much interest in the field of large-scale graph data processing, to gain an understanding of how Pregel-like systems perform, we conduct a study to experimentally compare Giraph, GPS, Mizan, and GraphLab on equal ground by considering graph and algorithm agnostic optimizations and by using several metrics

background on BSP and Pregel: pp.2 LHS, bulk synchronous parallel (BSP) is a parallel programming model with a message passing interface (MPI), Pregel is one of the first BSP implementations that provides a native API specifically for programming graph algorithms, graph computations are specified in terms of what each vertex has to compute; edges are communication channels for transmitting computation results from one vertex to another, and do not participate in the computation, the computation is split into supersteps, at each superstep a vertex can execute a user-defined function, send or receive messages to its neighbors, and change its state from active to inactive, to avoid communication overheads Pregel preserves data locality by ensuring computation is performed on locally stored data, as a result Pregel supports only graphs that fit in memory
pp.2 RHS, Pregel uses a master/workers model, each worker independently invokes a compute() function on the vertices in its portion of the graph, workers also maintain a message queue to receive messages from the vertices of other workers

systems tested: pp.3 LHS, large adjacency list partitioning (LALP) works by partitioning the adjacency lists of high-degree vertices across different workers; pp.3 RHS, dynamic migration repartitions the graph during computation by migrating vertices between workers, to improve workload balance and network usage

pp.4 LHS, unlike the previous systems GraphLab uses the GAS decomposition (Gather, Apply, Scatter), which is similar to but fundamentally different from the BSP model, in the GAS model a vertex accumulates information about its neighborhood in the Gather phase, applies the accumulated value in the Apply phase, and updates its adjacent vertices and edges and activates its neighboring vertices in the Scatter phase, in particular vertices can directly pull their neighbor's data (via Gather) without having to explicitly receive messages from those neighbors, in contrast a vertex under the BSP model can learn its neighbor's values only via the messages that its neighbors push to it, another key difference is that GraphLab partitions graphs using vertex cuts rather than edge cuts, which allow high degree vertices to be partitioned across multiple machines