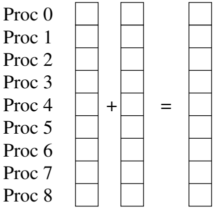
Parallel Algorithms

Granularity

An example: adding together two large vectors, maybe on shared memory, maybe on distributed memory

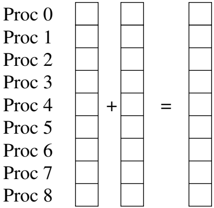


Adding vectors

Parallel Algorithms

Granularity

An example: adding together two large vectors, maybe on shared memory, maybe on distributed memory



Adding vectors

The simple fine grain allocation of one add per processor might not be the best if communications costs dictate otherwise

Parallel Algorithms

Granularity

For example, if the time it takes to get the data to the individual processors is large we would want to reduce the data movement

Parallel Algorithms

Granularity

For example, if the time it takes to get the data to the individual processors is large we would want to reduce the data movement

And in current memory architectures, it could take roughly the same amount of time to move one byte as it takes to move 10 or 100 or 1000 bytes

Parallel Algorithms

Granularity

For example, if the time it takes to get the data to the individual processors is large we would want to reduce the data movement

And in current memory architectures, it could take roughly the same amount of time to move one byte as it takes to move 10 or 100 or 1000 bytes

Time $=$ fixed overhead in setting up the transfer $+$ variable overhead in doing the transfer

Parallel Algorithms

Granularity

Thus: if we need to move data, move it in large chunks

Thus: if we need to move data, move it in large chunks

So, typically, we would have each processor would take a selection of elements and add them sequentially

Parallel Algorithms

Granularity

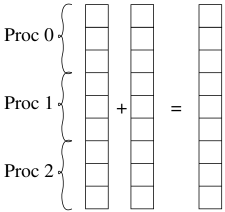
Thus: if we need to move data, move it in large chunks

So, typically, we would have each processor would take a selection of elements and add them sequentially

Larger grains of computation

Parallel Algorithms

Granularity

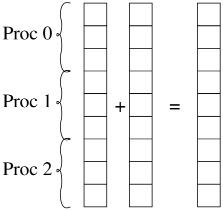


Adding contiguous blocks

They might be in contiguous chunks or spread somehow across the vectors, depending on the memory architecture

Parallel Algorithms

Granularity



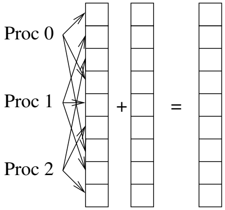
Adding contiguous blocks

They might be in contiguous chunks or spread somehow across the vectors, depending on the memory architecture

For example, CPUs like blocked data (0,1,2,3) (4,5,6,7) . . .

Parallel Algorithms

Granularity



Strided data

They might be in contiguous chunks or spread somehow across the vectors, depending on the memory architecture

For example, CPUs like blocked data (0,1,2,3) (4,5,6,7) . . . , while GPUs like strided data (0,4,8,12) (1,5,9,13) . . .

Parallel Algorithms

Divide and Conquer

The size of the grain we need will dictate the number of chunks we chop the problem into

Parallel Algorithms

Divide and Conquer

The size of the grain we need will dictate the number of chunks we chop the problem into

How many sub-problems should we have on each core?

The size of the grain we need will dictate the number of chunks we chop the problem into

How many sub-problems should we have on each core?

It is sometimes recommended that you have a “few” sub-problems per processor

The size of the grain we need will dictate the number of chunks we chop the problem into

How many sub-problems should we have on each core?

It is sometimes recommended that you have a “few” sub-problems per processor

This allows you to overlap communications with computation

Parallel Algorithms

Divide and Conquer

The size of the grain we need will dictate the number of chunks we chop the problem into

How many sub-problems should we have on each core?

It is sometimes recommended that you have a “few” sub-problems per processor

This allows you to overlap communications with computation

While a sub-problem is waiting for some data, the processor can continue computing on another sub-problem

How many is “a few”?

How many is “a few”?

It depends

How many is “a few”?

It depends

GPUs like to have *very many* many sub-problems per cores: as graphics problems need to push a lot of data around the processors would need to hang around doing nothing while waiting for data a lot: unless they have lots of other sub-problems to work on

Back to divide and conquer of adding numbers: isn't the merge step "add the values together" just another instance of the original question?

Back to divide and conquer of adding numbers: isn’t the merge step “add the values together” just another instance of the original question?

Yes, so a lot of divide and conquer methods are deeply recursive (not all, though)

Parallel Algorithms

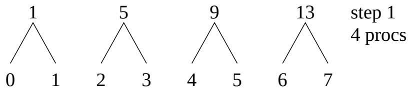
Divide and Conquer

This summation problem is usually regarded as

- if the number of values is small then
-     add them directly, sequentially
-     return the sum
- else divide them into two chunks
- recursively sum the parts in parallel
- add the two results
- return the sum

Parallel Algorithms

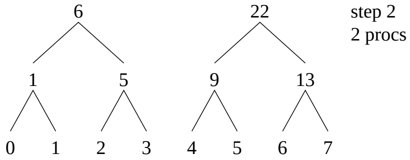
Divide and Conquer



Add pairs

Parallel Algorithms

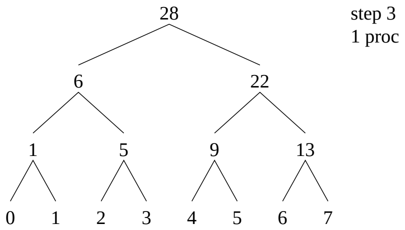
Divide and Conquer



Add pairs of sums

Parallel Algorithms

Divide and Conquer



Add final pair

We can compute the speedup and efficiency of this. We ignore communications overhead, so essentially we are using a PRAM model

Parallel Algorithms

Divide and Conquer

We can compute the speedup and efficiency of this. We ignore communications overhead, so essentially we are using a PRAM model

Time on a sequential processor: 7

We can compute the speedup and efficiency of this. We ignore communications overhead, so essentially we are using a PRAM model

Time on a sequential processor: 7

Time on this parallel system: 3

We can compute the speedup and efficiency of this. We ignore communications overhead, so essentially we are using a PRAM model

Time on a sequential processor: 7

Time on this parallel system: 3

Speedup $= 7/3 = 2.33$

Parallel Algorithms

Divide and Conquer

We can compute the speedup and efficiency of this. We ignore communications overhead, so essentially we are using a PRAM model

Time on a sequential processor: 7

Time on this parallel system: 3

Speedup $= 7/3 = 2.33$

Efficiency, using 4 processors: $2.33/4 = 58\%$

We can compute the speedup and efficiency of this. We ignore communications overhead, so essentially we are using a PRAM model

Time on a sequential processor: 7

Time on this parallel system: 3

Speedup $= 7/3 = 2.33$

Efficiency, using 4 processors: $2.33/4 = 58\%$

Note we are only using all the processors in the first step: thereafter there is increasing amounts of idle hardware

Divide and conquer is a good approach as long as you use it carefully

Divide and conquer is a good approach as long as you use it carefully

It is natural and easy to understand

Parallel Algorithms

Divide and Conquer

Divide and conquer is a good approach as long as you use it carefully

It is natural and easy to understand

It is fairly easy to program

Parallel Algorithms

Divide and Conquer

Divide and conquer is a good approach as long as you use it carefully

It is natural and easy to understand

It is fairly easy to program

It scales well to very large problems

Parallel Algorithms

Divide and Conquer

Divide and conquer is a good approach as long as you use it carefully

It is natural and easy to understand

It is fairly easy to program

It scales well to very large problems

But not all problems break up arbitrarily like this

Parallel Algorithms

Divide and Conquer

Divide and conquer is a good approach as long as you use it carefully

It is natural and easy to understand

It is fairly easy to program

It scales well to very large problems

But not all problems break up arbitrarily like this

And merging the parts can be as hard as the original problem

Parallel Algorithms

Divide and Conquer

It is a good technique to use in sequential systems, too

Parallel Algorithms

Divide and Conquer

It is a good technique to use in sequential systems, too

Recall merge sort (divide and conquer) is much better than bubble sort

Parallel Algorithms

Divide and Conquer

It is a good technique to use in sequential systems, too

Recall merge sort (divide and conquer) is much better than bubble sort

Bubble sort isn't parallelisable in any meaningful way (while still remaining essentially a bubble sort)

Parallel Algorithms

Divide and Conquer

It is a good technique to use in sequential systems, too

Recall merge sort (divide and conquer) is much better than bubble sort

Bubble sort isn't parallelisable in any meaningful way (while still remaining essentially a bubble sort)

The Fast Fourier Transform is a prime example of a good sequential application of divide and conquer

Of course splitting up isn't always the best option when you have a big problem. Counselling often works.
Anonymous. CM30225 exam, January 2011

Parallel Algorithms

Provider/Consumer

Terminology: we shall describe a method that previously was called “master/slave”: if you need to look it up, you will find it under this name

Parallel Algorithms

Provider/Consumer

Terminology: we shall describe a method that previously was called “master/slave”: if you need to look it up, you will find it under this name

Until a generally agreed replacement terminology is decided, we shall be calling it “provider/consumer”

Parallel Algorithms

Provider/Consumer

Divide and conquer is a way of arranging the problem. We now look at a way of arranging the control of the processing

Parallel Algorithms

Provider/Consumer

Divide and conquer is a way of arranging the problem. We now look at a way of arranging the control of the processing

Provider/consumer is a technique where there is a single main thread that determines what many consumer threads do

Parallel Algorithms

Provider/Consumer

Divide and conquer is a way of arranging the problem. We now look at a way of arranging the control of the processing

Provider/consumer is a technique where there is a single main thread that determines what many consumer threads do

For example, to do a large matrix multiplication, the main thread could get many consumer threads to do sub-parts of the operation

Parallel Algorithms

Provider/Consumer

Divide and conquer is a way of arranging the problem. We now look at a way of arranging the control of the processing

Provider/consumer is a technique where there is a single main thread that determines what many consumer threads do

For example, to do a large matrix multiplication, the main thread could get many consumer threads to do sub-parts of the operation

When the consumers are done the main thread can continue

Provider/consumer aligns naturally with divide and conquer, but usually not in a recursive way: in most uses the consumers don’t use sub-consumers

Parallel Algorithms

Provider/Consumer

Provider/consumer aligns naturally with divide and conquer, but usually not in a recursive way: in most uses the consumers don't use sub-consumers

Note: these ideas are not mutually exclusive, but they tend to overlap somewhat

Parallel Algorithms

Provider/Consumer

Provider/consumer is also related to the *server farm*, where a (large) collection of machines waits for problems to be sent to them

Parallel Algorithms

Provider/Consumer

Provider/consumer is also related to the *server farm*, where a (large) collection of machines waits for problems to be sent to them

For example, to do a search Google might send out sub-parts of the search to a collection of machines, and then collate the results

Provider/consumer is also related to the *server farm*, where a (large) collection of machines waits for problems to be sent to them

For example, to do a search Google might send out sub-parts of the search to a collection of machines, and then collate the results

In any case, in provider/consumer there is an asymmetry of control: one thread controlling several others

Parallel Algorithms

Manager/Worker

Provider/consumer is superficially quite similar to *manager/worker*, also called *bag of tasks*

Provider/consumer is superficially quite similar to *manager/worker*, also called *bag of tasks*

In this, there is a global set of problems to process held by the manager and the workers request a problem from the manager as they need

Provider/consumer is superficially quite similar to *manager/worker*, also called *bag of tasks*

In this, there is a global set of problems to process held by the manager and the workers request a problem from the manager as they need

A different control than provider/consumer

Provider/consumer is superficially quite similar to *manager/worker*, also called *bag of tasks*

In this, there is a global set of problems to process held by the manager and the workers request a problem from the manager as they need

A different control than provider/consumer

This allows easy *load balancing* on the workers

Parallel Algorithms

Load Balancing

Load balancing is one thing to do to approach a good efficiency

Parallel Algorithms

Load Balancing

Load balancing is one thing to do to approach a good efficiency

For example, if we have two big (time consuming) problems and two small ones, and two processors it makes sense to give each processor one big and one small

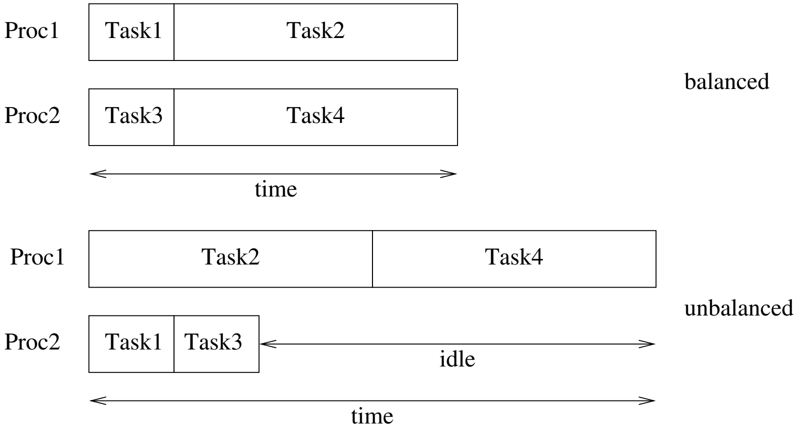
Load balancing is one thing to do to approach a good efficiency

For example, if we have two big (time consuming) problems and two small ones, and two processors it makes sense to give each processor one big and one small

If we give one processor both big problems and the other both the little ones it is clear our speedup and efficiency will both be lower as the second processor will soon be idling while we wait for the first to finish

Parallel Algorithms

Load Balancing



Balanced and unbalanced computations

Parallel Algorithms

Load Balancing

Load balancing tries to spread out the workload in a sensible fashion

Parallel Algorithms

Load Balancing

Load balancing tries to spread out the workload in a sensible fashion

It requires us to have some idea of how big each sub-problem is, namely a good estimate of their granularity

Parallel Algorithms

Load Balancing

Load balancing tries to spread out the workload in a sensible fashion

It requires us to have some idea of how big each sub-problem is, namely a good estimate of their granularity

Theory tells us that this is impossible in general, but for the most part in practice we can make a decent guess

Parallel Algorithms

Load Balancing

Load balancing tries to spread out the workload in a sensible fashion

It requires us to have some idea of how big each sub-problem is, namely a good estimate of their granularity

Theory tells us that this is impossible in general, but for the most part in practice we can make a decent guess

Many large problems are quite regular in structure and as so fairly amenable to this kind of analysis, but there are many irregular problems that are not so easy

And even if we have a good idea of the size of each task, finding an even balance can be difficult (the *multiprocessor scheduling problem* is NP-hard)

Parallel Algorithms

Load Balancing

And even if we have a good idea of the size of each task, finding an even balance can be difficult (the *multiprocessor scheduling problem* is NP-hard)

Note that load balancing applies to more than just CPU cycles: there's memory, network bandwidth and any other limited resource

Parallel Algorithms

Load Balancing

And even if we have a good idea of the size of each task, finding an even balance can be difficult (the *multiprocessor scheduling problem* is NP-hard)

Note that load balancing applies to more than just CPU cycles: there's memory, network bandwidth and any other limited resource

And these play off against each other: it may be worthwhile to put two sub-problems on the same processor if they need to swap data and this will reduce communications overheads

Parallel Algorithms

Load Balancing

And even if we have a good idea of the size of each task, finding an even balance can be difficult (the *multiprocessor scheduling problem* is NP-hard)

Note that load balancing applies to more than just CPU cycles: there's memory, network bandwidth and any other limited resource

And these play off against each other: it may be worthwhile to put two sub-problems on the same processor if they need to swap data and this will reduce communications overheads

Load balancing is quite similar to process scheduling in operating systems: but now we might be working with large distributed systems

Parallel Algorithms

Manager/Worker

The manager/worker model is good because it is somewhat self-balancing on average

Parallel Algorithms

Manager/Worker

The manager/worker model is good because it is somewhat self-balancing on average

A worker that happens to get a small task will soon be back for another task

Parallel Algorithms

Manager/Worker

The manager/worker model is good because it is somewhat self-balancing on average

A worker that happens to get a small task will soon be back for another task

Provider/consumer might have to take some care over which tasks it supplies to where

Parallel Algorithms

Manager/Worker

The manager/worker model is good because it is somewhat self-balancing on average

A worker that happens to get a small task will soon be back for another task

Provider/consumer might have to take some care over which tasks it supplies to where

Though this is not a problem if all sub-tasks are the same size. Provider/consumer is good for this case and might be simpler to implement than manager/worker

Parallel Algorithms

Thread Pools

A way of implementing manager/worker is to use thread pools

Parallel Algorithms

Thread Pools

A way of implementing manager/worker is to use thread pools

We have a pool of threads that take tasks from one or more managers

Parallel Algorithms

Thread Pools

A way of implementing manager/worker is to use thread pools

We have a pool of threads that take tasks from one or more managers

After each task, a thread goes back to the manager for a new task

Parallel Algorithms

Thread Pools

A way of implementing manager/worker is to use thread pools

We have a pool of threads that take tasks from one or more managers

After each task, a thread goes back to the manager for a new task

We mitigate the overhead of thread creation/deletion

Parallel Algorithms

Thread Pools

A way of implementing manager/worker is to use thread pools

We have a pool of threads that take tasks from one or more managers

After each task, a thread goes back to the manager for a new task

We mitigate the overhead of thread creation/deletion

The thread pool can be managed within the program, or system-wide by the OS

If the pool is managed by the operating system it can have a global view of how the entire system's resources are being used

Parallel Algorithms

Thread Pools

If the pool is managed by the operating system it can have a global view of how the entire system's resources are being used

Threads can be passed to any program, again reducing the overall overheads

Parallel Algorithms

Thread Pools

If the pool is managed by the operating system it can have a global view of how the entire system's resources are being used

Threads can be passed to any program, again reducing the overall overheads

And the OS can increase or decrease the number of threads according to how the whole system is loaded

Parallel Algorithms

Thread Pools

If the pool is managed by the operating system it can have a global view of how the entire system's resources are being used

Threads can be passed to any program, again reducing the overall overheads

And the OS can increase or decrease the number of threads according to how the whole system is loaded

This requires OS support, of course: think of the issues of access to the program's address space by each thread

Parallel Algorithms

Thread Pools: GCD

This is the idea of Apple's "solution" to parallelism: *Grand Central Dispatch* (GCD)

Parallel Algorithms

Thread Pools: GCD

This is the idea of Apple's "solution" to parallelism: *Grand Central Dispatch* (GCD)

Rather than programs creating their own threads, e.g., using pthreads, they use (and re-use) the OS's threads from a global thread pool

Parallel Algorithms

Thread Pools: GCD

This is the idea of Apple's "solution" to parallelism: *Grand Central Dispatch* (GCD)

Rather than programs creating their own threads, e.g., using pthreads, they use (and re-use) the OS's threads from a global thread pool

A program gets access to a pool thread by putting a task, e.g., a function call, on a *queue*

Parallel Algorithms

Thread Pools: GCD

This is the idea of Apple's "solution" to parallelism: *Grand Central Dispatch* (GCD)

Rather than programs creating their own threads, e.g., using pthreads, they use (and re-use) the OS's threads from a global thread pool

A program gets access to a pool thread by putting a task, e.g., a function call, on a *queue*

The worker threads pick tasks off the queues and execute them

Parallel Algorithms

Thread Pools: GCD

This is the idea of Apple's "solution" to parallelism: *Grand Central Dispatch* (GCD)

Rather than programs creating their own threads, e.g., using pthreads, they use (and re-use) the OS's threads from a global thread pool

A program gets access to a pool thread by putting a task, e.g., a function call, on a *queue*

The worker threads pick tasks off the queues and execute them

Parallelism is obtained by having lots of worker threads taking tasks

Parallel Algorithms

Thread Pools: GCD

So GCD gets the automatic load balancing of manager/worker

Parallel Algorithms

Thread Pools: GCD

So GCD gets the automatic load balancing of manager/worker

GCD can also provide mutual exclusion

So GCD gets the automatic load balancing of manager/worker

GCD can also provide mutual exclusion

By creating and using a special queue called a *serial queue* a program indicates it wants just one thread to service this new queue

So GCD gets the automatic load balancing of manager/worker

GCD can also provide mutual exclusion

By creating and using a special queue called a *serial queue* a program indicates it wants just one thread to service this new queue

As only one thread executes tasks from this queue there can be no issues of interference between threads on that queue

Parallel Algorithms

Thread Pools: GCD

So, roughly speaking, code like

```
            fblock = make_lock();
get_lock(fblock);          get_lock(fblock);
foo();                     bar();
free_lock(fblock);         free_lock(fblock);
```

Parallel Algorithms

Thread Pools: GCD

So, roughly speaking, code like

```
            fblock = make_lock();
get_lock(fblock);          get_lock(fblock);
foo();                     bar();
free_lock(fblock);         free_lock(fblock);
```

becomes

So, roughly speaking, code like

```
            fblock = make_lock();
get_lock(fblock);          get_lock(fblock);
foo();                     bar();
free_lock(fblock);         free_lock(fblock);
```

becomes

```
          fbqueue = make_serial_queue();
enqueue(foo, fbqueue);     enqueue(bar, fbqueue);
```

Parallel Algorithms

Thread Pools: GCD

There is no loss of parallelism by using a single thread to process the queue in this case, as the critical region has to be serialised anyway

Parallel Algorithms

Thread Pools: GCD

There is no loss of parallelism by using a single thread to process the queue in this case, as the critical region has to be serialised anyway

Though you do need to be careful about making the function called as small as possible, for the usual reasons

Parallel Algorithms

Thread Pools: GCD

There is no loss of parallelism by using a single thread to process the queue in this case, as the critical region has to be serialised anyway

Though you do need to be careful about making the function called as small as possible, for the usual reasons

Just as each critical resource needs its own lock, in GCD each critical resource needs its own serial queue

Parallel Algorithms

Thread Pools: GCD

There is no loss of parallelism by using a single thread to process the queue in this case, as the critical region has to be serialised anyway

Though you do need to be careful about making the function called as small as possible, for the usual reasons

Just as each critical resource needs its own lock, in GCD each critical resource needs its own serial queue

If a resource would need *two* locks, then you need two queues and put a function on the first queue that itself puts another function on the second queue that actually executes the required critical region

Parallel Algorithms

Thread Pools: GCD

There is no loss of parallelism by using a single thread to process the queue in this case, as the critical region has to be serialised anyway

Though you do need to be careful about making the function called as small as possible, for the usual reasons

Just as each critical resource needs its own lock, in GCD each critical resource needs its own serial queue

If a resource would need *two* locks, then you need two queues and put a function on the first queue that itself puts another function on the second queue that actually executes the required critical region

Somewhat fiddly

Parallel Algorithms

Thread Pools: GCD

Rather than placing functions in queues, Apple's implementation makes extensive use of *closures*, a feature they have added to their version of C

Parallel Algorithms

Thread Pools: GCD

Rather than placing functions in queues, Apple's implementation makes extensive use of *closures*, a feature they have added to their version of C

They call them *blocks*, but they are similar to lambdas in other languages

Parallel Algorithms

Thread Pools: GCD

Rather than placing functions in queues, Apple's implementation makes extensive use of *closures*, a feature they have added to their version of C

They call them *blocks*, but they are similar to lambdas in other languages

Of course, closures were imported from the functional programming style: as long as we have referential transparency the individual tasks can run completely independently

Apple’s claim is that queues are cheap to create and use, while threads and mutexes are expensive

Parallel Algorithms

Thread Pools: GCD

Apple's claim is that queues are cheap to create and use, while threads and mutexes are expensive

They are less effusive on costs like mutual exclusion on the queue itself; costs of the OS deciding on which thread services which queue; costs of the virtual address mapping of the threads as they get assigned to processes; cost of creation and manipulation of closures; and so on

Parallel Algorithms

Thread Pools: GCD

Apple's claim is that queues are cheap to create and use, while threads and mutexes are expensive

They are less effusive on costs like mutual exclusion on the queue itself; costs of the OS deciding on which thread services which queue; costs of the virtual address mapping of the threads as they get assigned to processes; cost of creation and manipulation of closures; and so on

We are still waiting to see if the GCD paradigm is easy to use in real programs or not!

While GCD uses thread pools at the OS level, the approach of a program implementing its own pool is quite common

Parallel Algorithms

Thread Pools

While GCD uses thread pools at the OS level, the approach of a program implementing its own pool is quite common

Again, and this is true for all these concurrency paradigms, this only works well if your problem happens to fit well into the pool or manager/worker patterns

Parallel Algorithms

Thread Pools

While GCD uses thread pools at the OS level, the approach of a program implementing its own pool is quite common

Again, and this is true for all these concurrency paradigms, this only works well if your problem happens to fit well into the pool or manager/worker patterns

One of the many issues encountered when designing parallel programs is choosing the right parallelism pattern

Exercise There is a Linux library `libdispatch` that implements (per process) GCD. Write some programs using it

Exercise There is a Linux library `libdispatch` that implements (per process) GCD. Write some programs using it

> *Disadvantages include that it is managed by Apple. No more needs to be said.*
> Anon, Jan 2023 CM30225 exam