Parallel Dot Product

https://github.com/VeriNum/pardotprod

Andrew W. Appel

Princeton
University

August 2022
narration put into writing, May 2025

Abstract

We demonstrate how to use the Verified Software Toolchain to prove correctness of a parallel program that uses barrier synchronization implemented with binary semaphores.

Verified
Software
Toolchain

That is: master thread breaks the job into T separate tasks, hands off T-1 tasks to other threads (that are already waiting for them) by releasing per-thread semaphores. Then the master works on one task, then waits on T-1 semaphores for all the other threads to finish their task.

As an example, we demonstrate parallel dot product. But the work-splitting API is quite general. It would easily apply to more useful parallel "clients" such as blocked matrix multiply or other algorithms that can use barrier synchronization.

Part 1: The client program and API

In part 1 we don't use formal methods at all. We just present the C program that we intend to prove correct. The program is divided into

- a parallelism API (for handling work that's split into T tasks) and
- an application "client" program that uses the API (in this case, parallel dot-product).

Simple Task Parallelism

All ranges are [lo,hi)

- Have a function to compute on big data

$$\sum_{i=0}^{n} x_i \cdot y_i$$

- Have T processors

- Divide computation into T subfunctions (compute in parallel)

$$\delta_t \stackrel{\text{def}}{=} \left\lfloor \frac{nt}{T} \right\rfloor \qquad \sum_{i=\delta_t}^{\delta_{t+1}} x_i \cdot y_i$$

- Combine subresults together

$$\sum_{t=0}^{T} \sum_{i=\delta_t}^{\delta_{t+1}} x_i \cdot y_i$$

API for work-splitting

parsplit.h

```
struct task *make_tasks (unsigned T);

void initialize_task (struct task *tasks,
                      unsigned t,
                      void (*f)(void *),
                      void *closure);

void do_tasks (struct task *tasks, unsigned T);
```

Start T-1 threads
(plus parent makes T)

Tell the t^{th} thread where to find its work

Run all the threads on their work
(can do this again and again)

What's in a **`struct task`** is the private information of the task-scheduling system (parsplit.c) but you can imagine it contains a couple of semaphores, among other things.

The client decides what operation is to be performed in a task, and passes that into `initialize_task` as the function-parameter f accompanied by supplementary client-side information called `closure`. The client will call `initialize_task` T times, with `t` ranging from 0 to T-1, presumably with different `closure` values for each one.

Scenario

Suppose you have T processors, and your program is going to compute many dot products on vectors of length n.

First, use `make_tasks` to create T threads, and then use `initialize_task` to tell each thread what work it's going to have to do.

```
void make_dotprod_tasks(unsigned T) {
  unsigned t;
  tasks = make_tasks(T);

   . . . /* more to come */ . . .

  for (t=0; t<T; t++)
        initialize_task(tasks, t,  ... /* more to come */ ... );
}
```

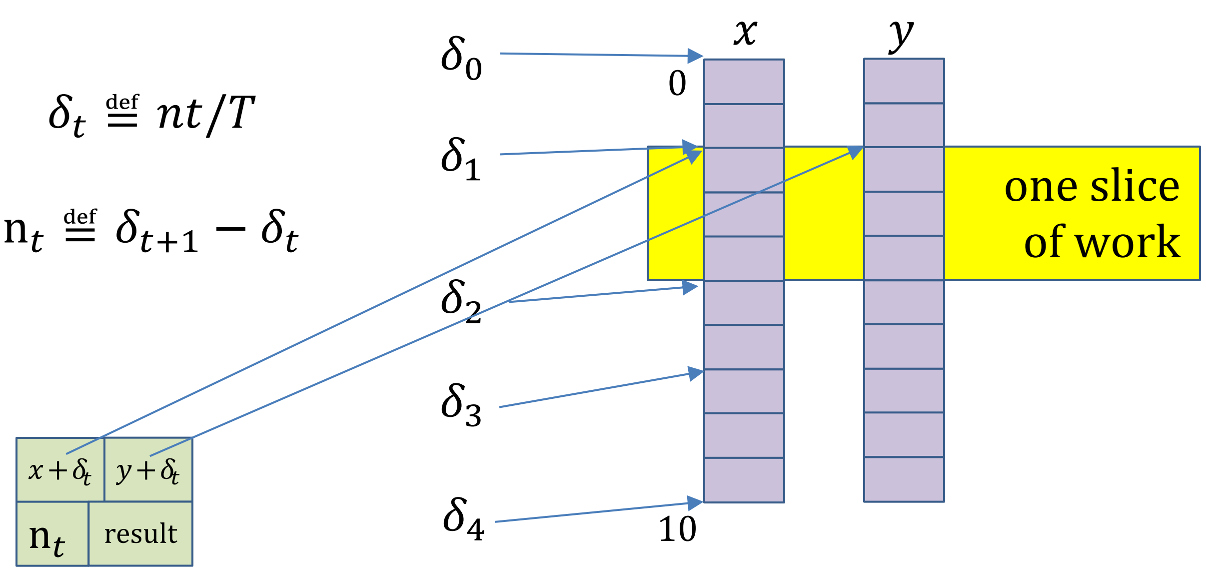
Later, when the program wants to compute a dot-product, it will update the per-task data (i.e., the vectors to be multiplied) and call `do_tasks` to start all T threads working.

Example: n=10, T=4

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```

$$\delta_t \stackrel{\text{def}}{=} nt/T$$

$$\mathrm{n}_t \stackrel{\text{def}}{=} \delta_{t+1} - \delta_t$$



Task description (`struct dotprod_task`)

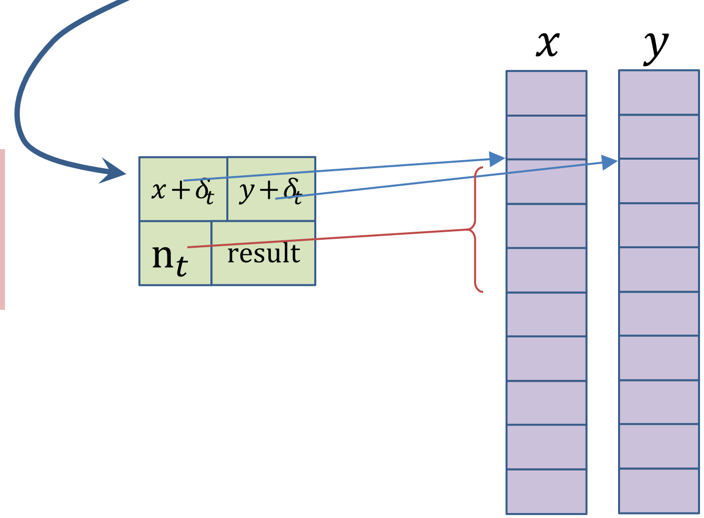
For the dot-product client, a `closure` (i.e., task-description) has pointers to vector-slice $x + \delta_t$, vector-slice $y + \delta_t$, length of the vector slices n_t, and a space into which the slice result can be written. If the original vector length is not exactly a multiple of the number of threads, then some slices will be a bit longer than others.

Application-specific subtask function

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```

```
void dotprod_worker(void *closure) {
  struct dotprod_task *w = closure;
  double result=0.0;
  unsigned n = w->n;
  double *vec1= w->vec1, *vec2= w->vec2;
  unsigned i;
  for (i=0; i<n; i++)
     result += vec1[i]*vec2[i];
  w->result=result;
}
```

$$\sum x_i \cdot y_i$$

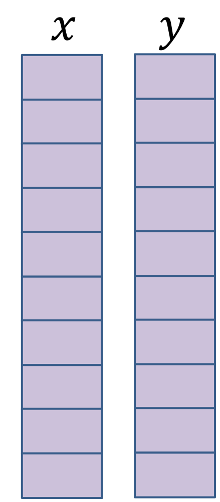


The **`dotprod_worker`** is given a pointer to a `closure` and computes the slice dot product for that task-description. The formal parameter has type `void*` instead of `struct dotprod_task *` because the task-scheduler (parsplit.c, parsplit.h) must be general enough that it doesn't even know the type of the task-description. Hence the assignment from `closure` to `w` in the first line of the function body.

How the client uses the API

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```

Suppose you want to compute the dot-product of vectors x and y, in parallel. Each vector has length n.




```
double dotprod(double *vec1, double *vec2, unsigned n) {






}
```

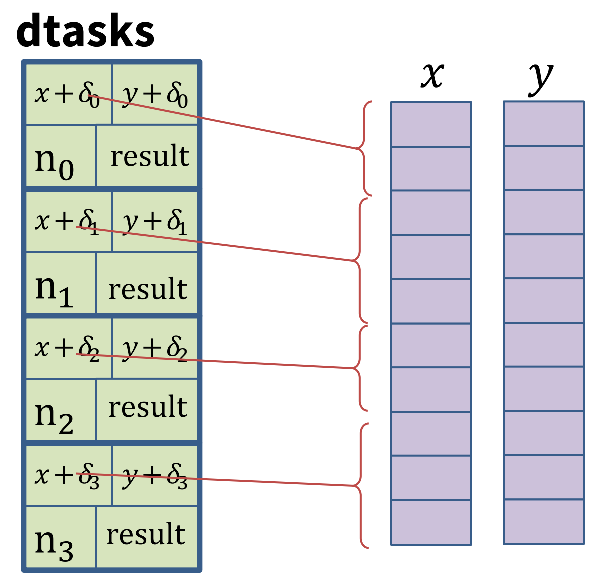
Creating the task-descriptions

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```

First, compute how you're going to break it into T tasks. If n is not an exact multiple of T, then the sizes n_0, n_1, n_2, ... won't be exactly the same.

$$\delta_t \stackrel{\text{def}}{=} nt/T$$

$$\mathrm{n}_t \stackrel{\text{def}}{=} \delta_{t+1} - \delta_t$$



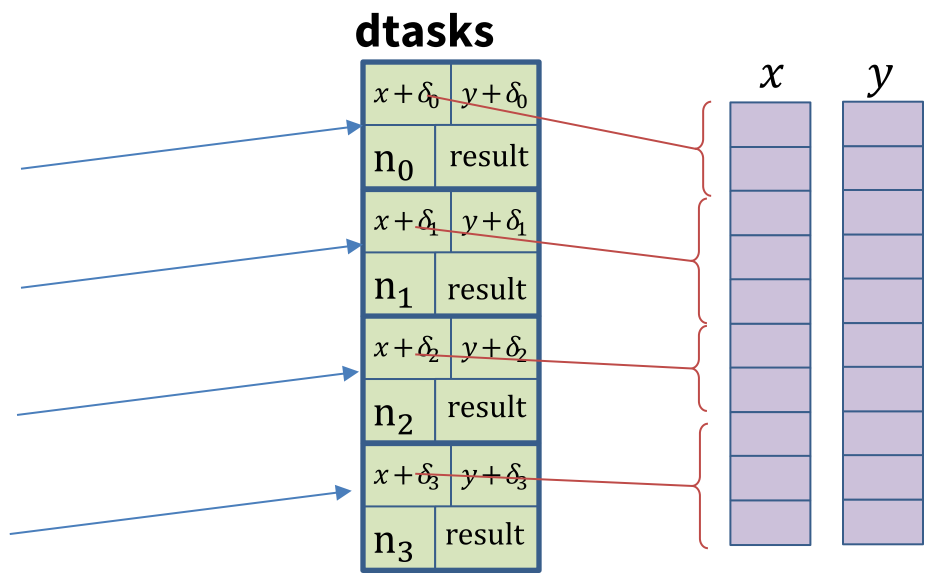

```
double dotprod(double *vec1, double *vec2, unsigned n) {
for (delta=0, t=0;  t<T;  t++) {
   dtasks[t].vec1=vec1+delta;
   dtasks[t].vec2=vec2+delta;
   delta_next = (t+1)*n/T;
   dtasks[t].n= delta_next-delta;
   delta=delta_next;
 }
  . . .
}
```

(For how & when the dtasks array was created, wait a couple of slides. Here we're just filling it in.)

dotprod_worker

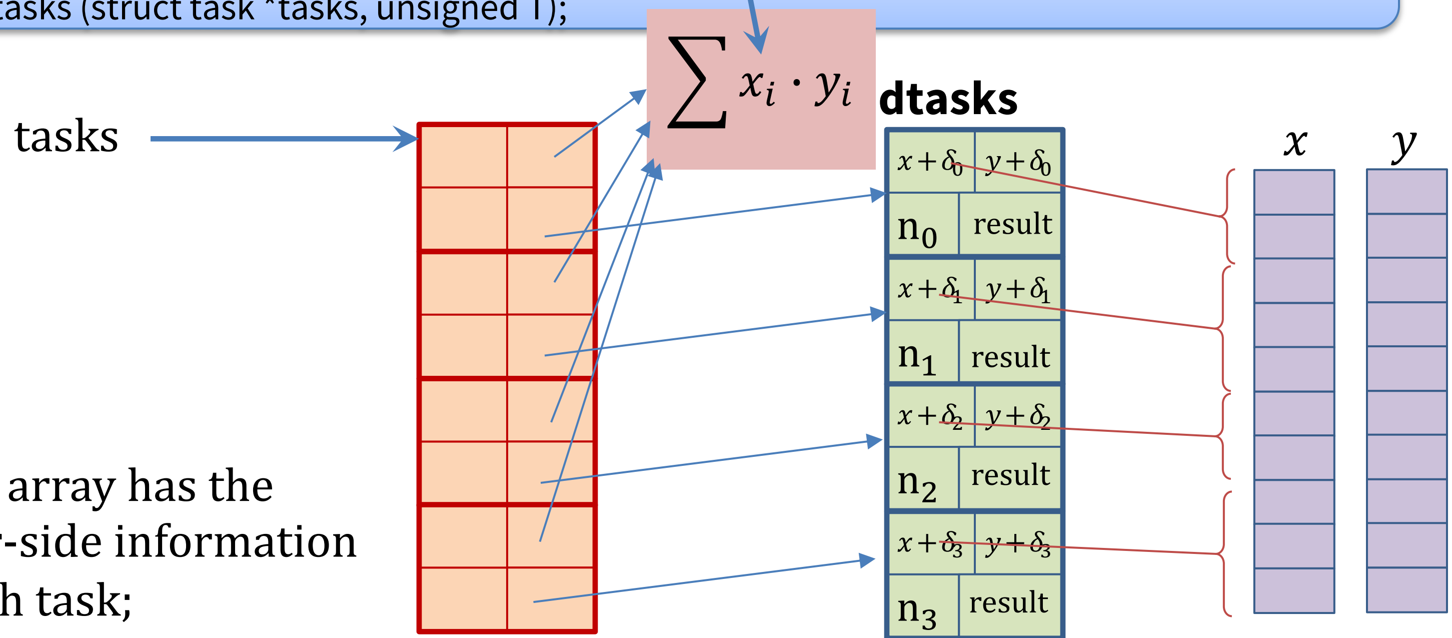
```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```

Each of these pointers is a "closure" that can be passed to the dotprod_worker function. The next step is to register each of these $n = 4$ closures with the task manager



do_tasks( )

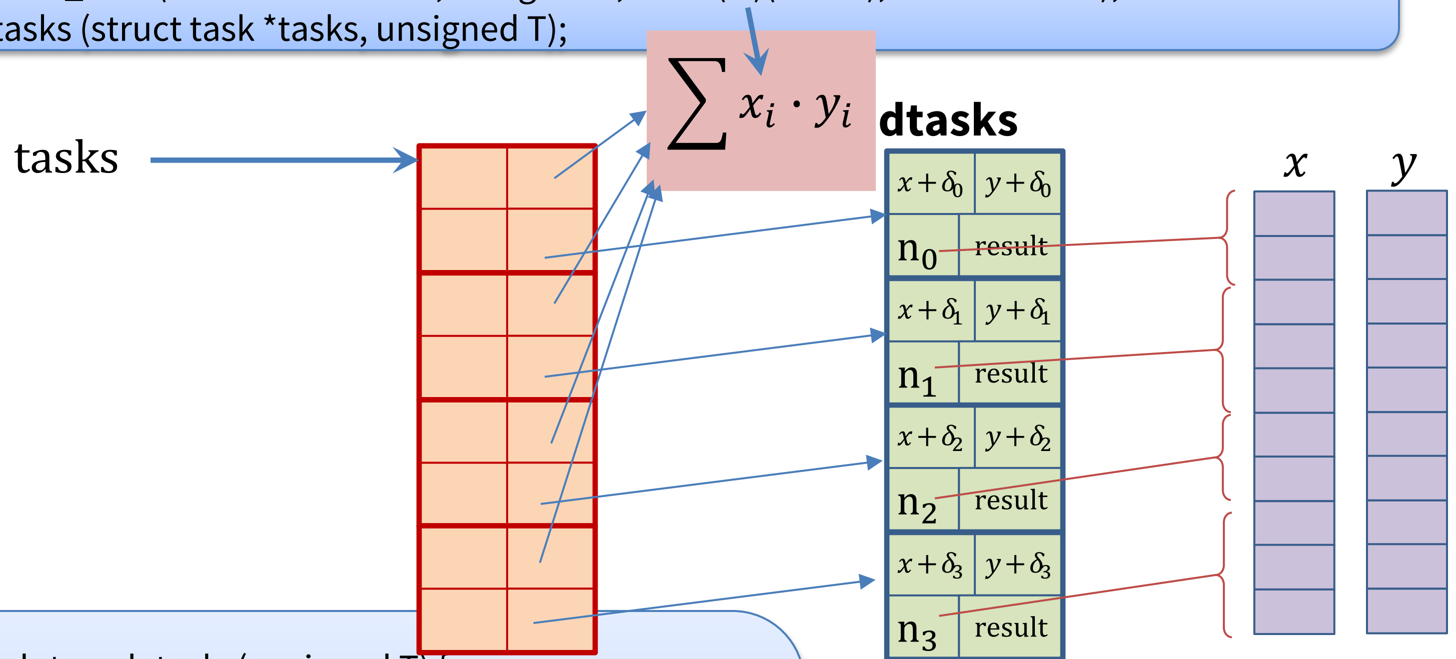
```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```



The tasks array has the scheduler-side information about each task; that includes, as shown here, pointer to what function f to execute and the specific closure information for each task.

make_tasks( ), initialize_task( )

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```



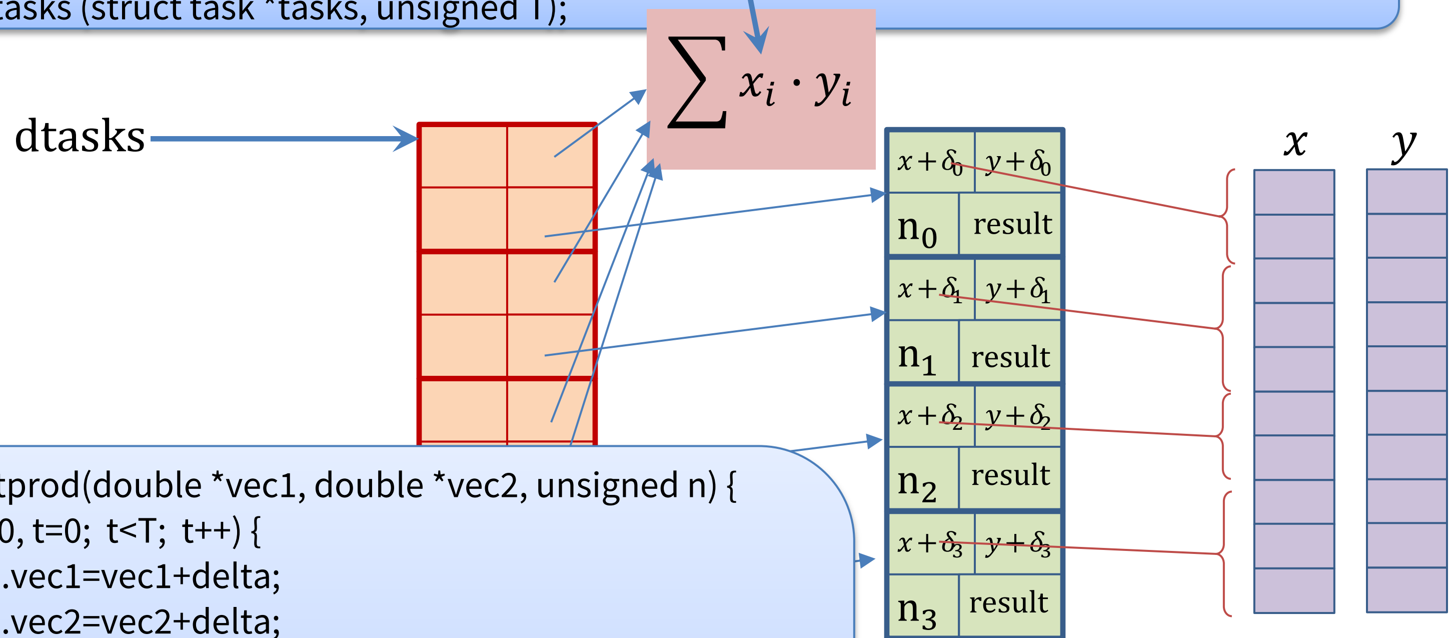

```
void make_dotprod_tasks(unsigned T) {
  unsigned t;
  tasks = make_tasks(T);
  num_threads=T;
  dtasks=(struct dotprod_task *)malloc(T*sizeof(…));
  for (t=0; t<T; t++)
        initialize_task(tasks, t, dotprod_worker, dtasks+t);
}
```

And here is the client's function that creates both the tasks array and the dtasks array, and fills in the tasks.

`num_threads` is a client-side global variable.

do_tasks( )

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```




```
double dotprod(double *vec1, double *vec2, unsigned n) {
for (delta=0, t=0;  t<T;  t++) {
   dtasks[t].vec1=vec1+delta;
   dtasks[t].vec2=vec2+delta;
   delta_next = (t+1)*n/T;
   dtasks[t].n= delta_next-delta;
   delta=delta_next;
 }
 do_tasks(tasks, T);
 for (result=0.0, t=0; t<T; t++)    result += dtasks[t].result;
 return result;
}
```

To compute a dot-product,

- update the per-task info (shown previously);
- call **`do_tasks`** to run the parallel jobs;
- collect all the results

That's the entire dotprod.c client

```
void dotprod_worker(void *closure) {
  struct dotprod_task *w = closure;
  double result
  unsigned n =
  double *vec1
>vec2;
  unsigned i;
  for (i=0; i<n; i
     result += ve
  w->result=resu
}
```

```
void make_dotprod_tasks(unsigned T) {
  unsigned t;
  tasks = make_tasks(T);
  num_threads=T;
  dtasks=(struct dotprod_task *)malloc(T*sizeof(…));
  for (t=0; t<T; t++)
        initialize_task(tasks, t, dotprod_worker, dtasks+t);
}
```

```
double dotprod(double *vec1, double *vec2, unsigned n) {
for (delta=0, t=0;  t<T;  t++) {
                ec1=vec1+delta;
                c2=vec2+delta;
                = (t+1)*n/T;
                 delta_next-delta;
                _next;

                ks, T);
               .0, t=0; t<T; t++)    result += dtasks[t].result;
 return result;
}
```

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```

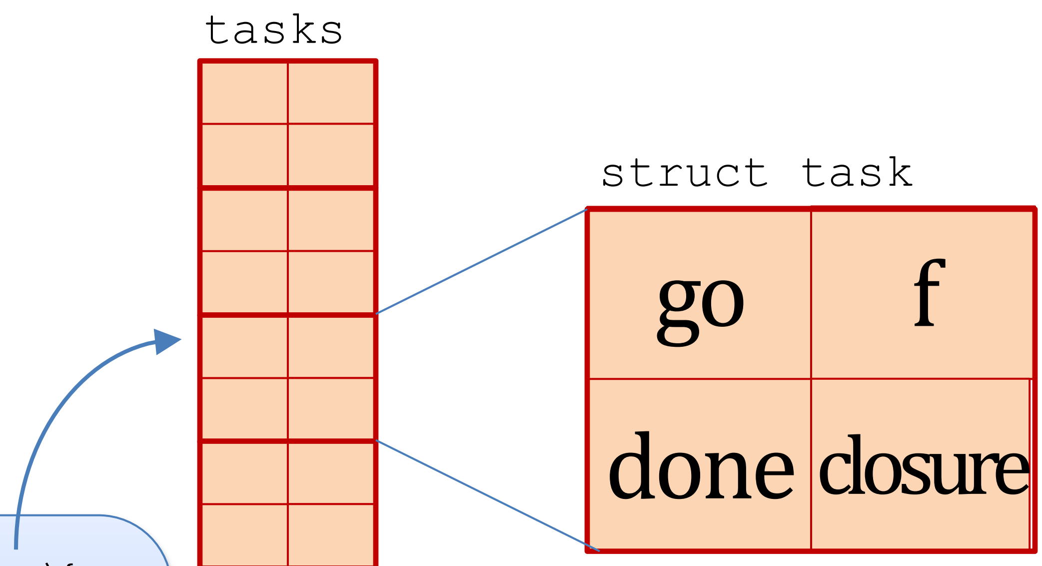
Part 2: How the parallelism is implemented

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```

This is the API; how do these functions work?

thread_worker( ), make_tasks( )

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```




```
int thread_worker(void *arg) {
  struct task *t = (struct task *)arg;
  while (1)  {
    acquire(t->go);
    t->f(t->closure);
    release(t->done);
   }
}
```

What runs in each thread is simple:

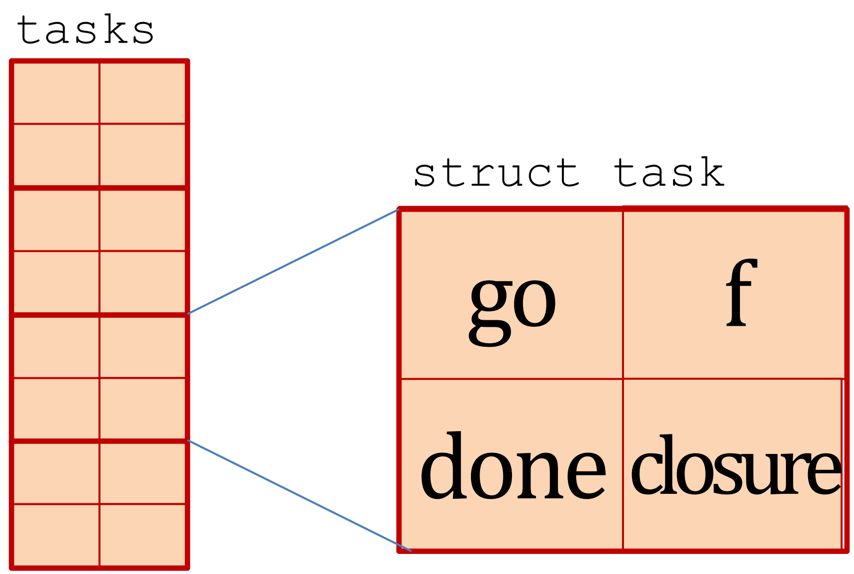
- wait for the **go** signal,
- run the function f on the closure,
- send the **done** signal.

Repeat forever.

thread_worker( ), make_tasks( )

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```

```
struct task *make_tasks(unsigned T) {
  tasks = malloc(T * sizeof (…));
  for (i=1; i<T; i++) {
    struct task *t = tasks+i;
    t->go = makelock();
    t->done = makelock();
    spawn(thread_worker, t);
  }
  return tasks;
}
```




```
int thread_worker(void *arg) {
  struct task *t = (struct task *)arg;
  while (1)  {
    acquire(t->go);
    t->f(t->closure);
    release(t->done);
   }
}
```

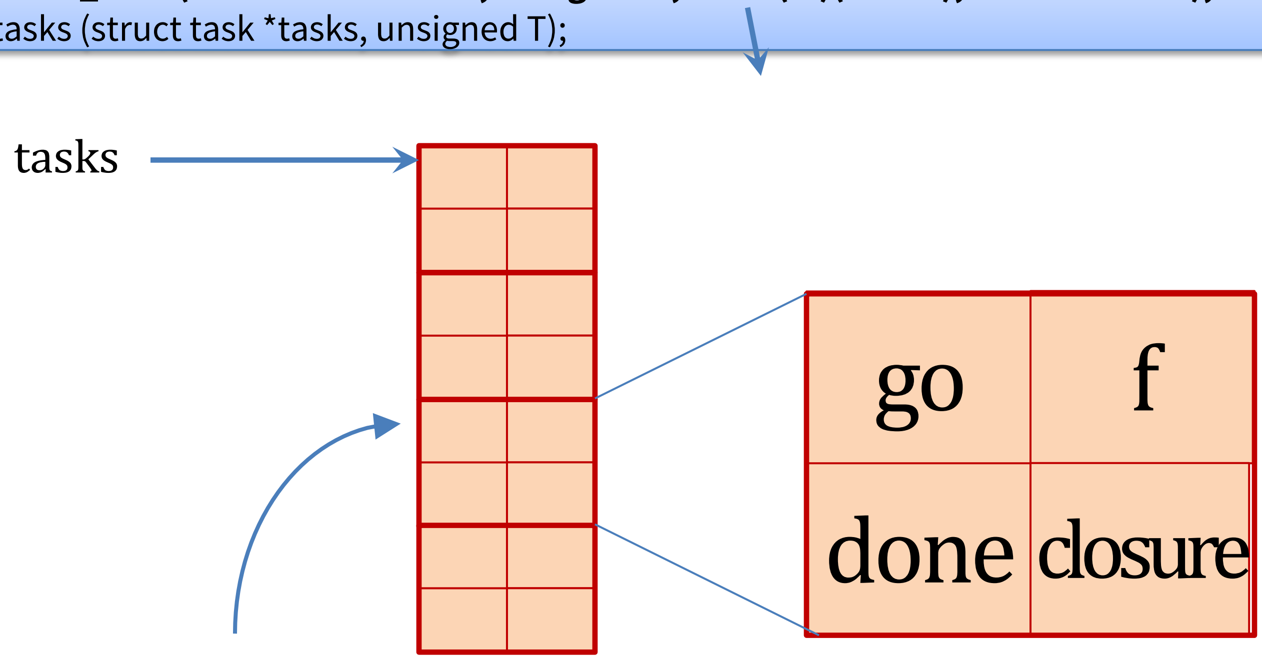
All **`make_tasks`** does is,

- create all the **go** and **done** semaphores
- spawn all the threads to run **`thread_worker`**

(semaphores are created in the *locked* state, so the first thing all those threads do is block on the acquire).

initialize_task ( )

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```

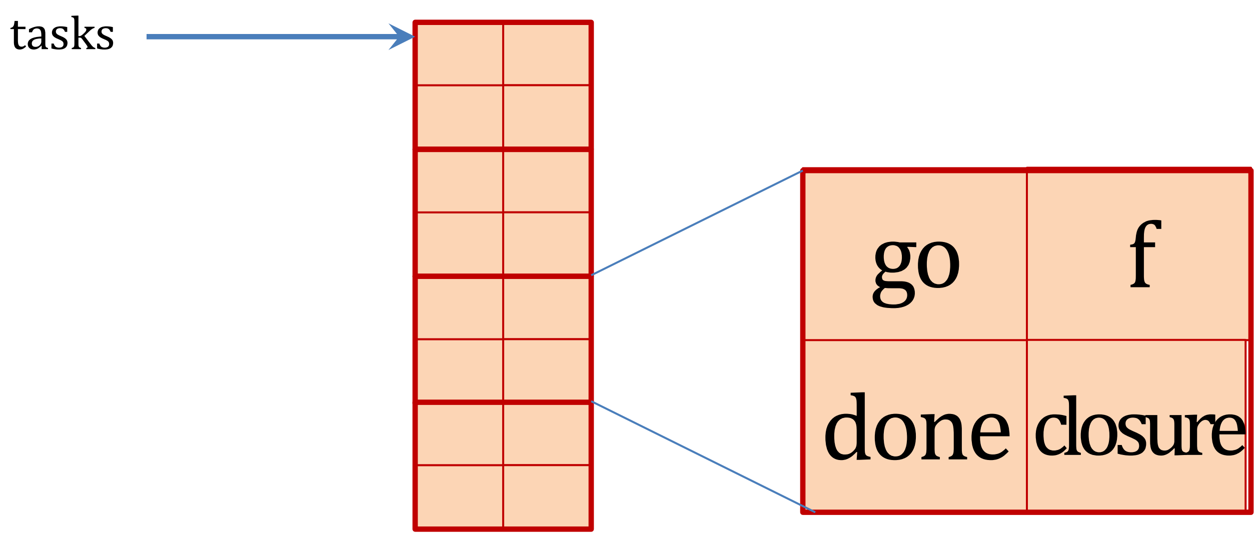



```
void initialize_task (struct task *tasks,
                      unsigned i,
                      void (*f)(void *),
                      void *closure) {
 tasks[i].f=f;
 tasks[i].closure=closure;
}
```

After the client calls **`make_tasks`** to spawn all the threads, then the client calls **`initialize_task`** to fill in the info about the function and closure.

do_tasks ( )

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```




```
void do_tasks(struct task *tasks, unsigned T) {
for (i=1; i<T; i++)
   release (tasks[i].go);
 tasks[0].f(tasks[0].closure);
 for (i=1; i<T; i++)
   acquire (tasks[i].done);
}
```

To compute a parallel dot product, the client calls `do_tasks` which is very simple: send each thread the **go** signal, compute task 0 locally, wait for each thread to send the **done** signal.

That's the entire parsplit.c

```
struct task *make_tasks (unsigned T);
void initialize_task (struct task *tasks, unsigned t,  void (*f)(void *), void *closure);
void do_tasks (struct task *tasks, unsigned T);
```

```
struct task *make_tasks(unsigne
  tasks = malloc(T * sizeof (…));
  for (i=1; i<T; i++) {
    struct task *t = tasks+i;
    t->go = makelock();
    t->done = makelock();
    spawn(thread_worker, t);
  }
  return tasks;
}
```

```
int thread_worker(void *arg) {
  struct task *t = (struct task *)arg;
  while (1)  {
    acquire(t->go);
    t->f(t->closure);
    release(t->done);
  }
}
```

```
void initialize_task (struct task *tasks,
      unsigned i, void (*f)(void *), void *closure) {
  tasks[i].f=f;
  tasks[i].closure=closure;
}
```

```
void do_tasks(struct task *tasks, unsigned T) {
for (i=1; i<T; i++)
    release (tasks[i].go);
  tasks[0].f(tasks[0].closure);
  for (i=1; i<T; i++)
    acquire (tasks[i].done);
}
```

Part 3

HOW TO PROVE IT

Separation Logic

If state satisfies the precondition

then it's safe to run the command

and the state after will satisfy the postcondition

{Pre} command {Post}

separating conjunction

$P * Q$ $\qquad\qquad$ $P' * Q$

Separation Logic

If state satisfies the precondition

then it's safe to run the command

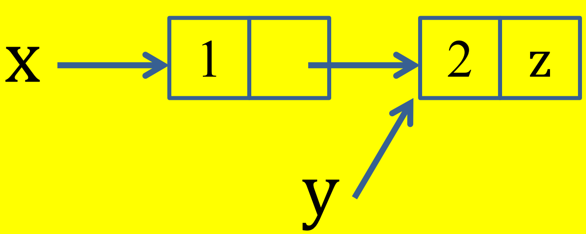
and the state after will satisfy the postcondition

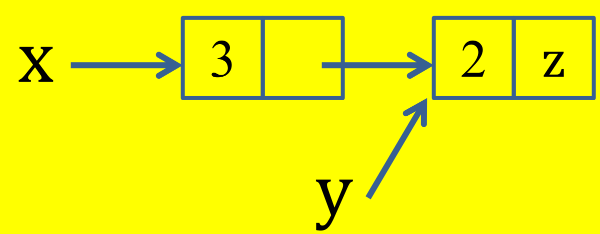
$$\{\text{Pre}\}\ \ \text{command}\ \ \{\text{Post}\}$$

$$P * Q \qquad\qquad P' * Q$$

separating conjunction

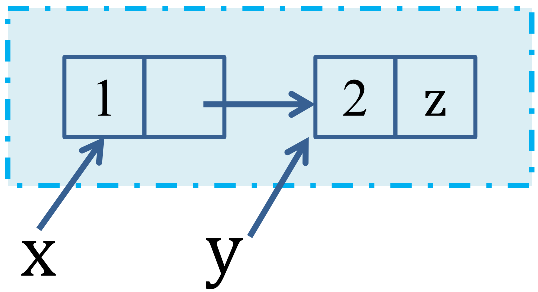
$$\{\ x \mapsto (1,y) * y \mapsto (2,z)\ \}\quad x.\text{data}=3;\quad \{\ x \mapsto (3,y) * y \mapsto (2,z)\ \}$$





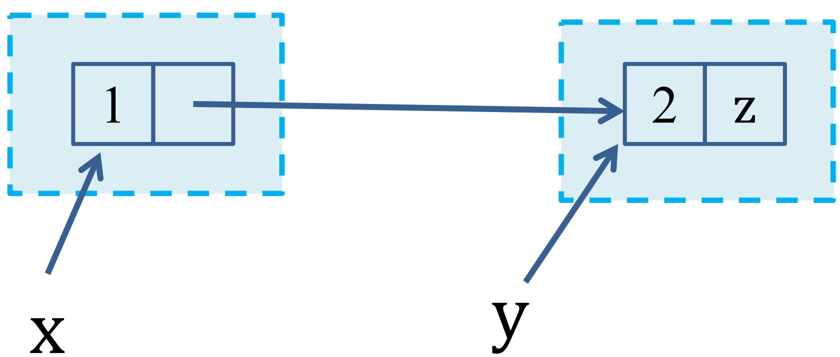
Heaplets in Separation Logic

$x \mapsto (1,y) * y \mapsto (2,z)$



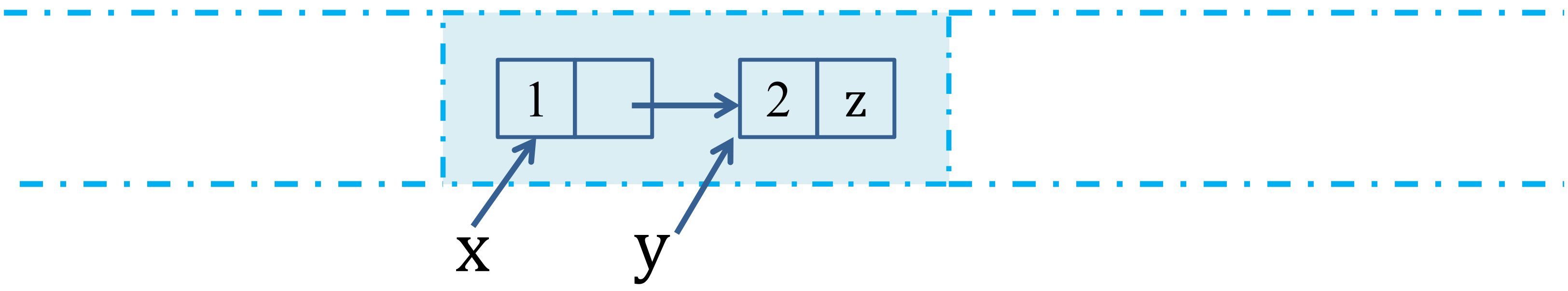
A "heaplet" is a model of a separating conjunct; it's a (not necessarily contiguous) part of memory with a given footprint (domain)

$x \mapsto (1,y) \qquad * \qquad y \mapsto (2,z)$

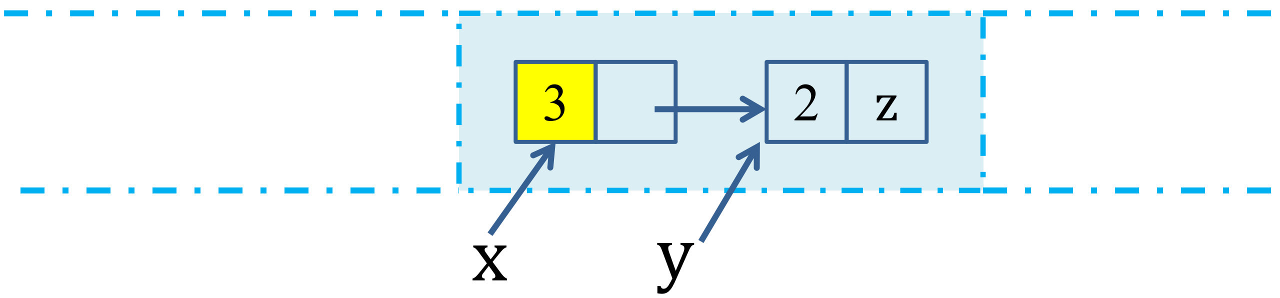


The separating conjunction $*$ is about the union of two disjoint footprints

Heaplets in Separation Logic



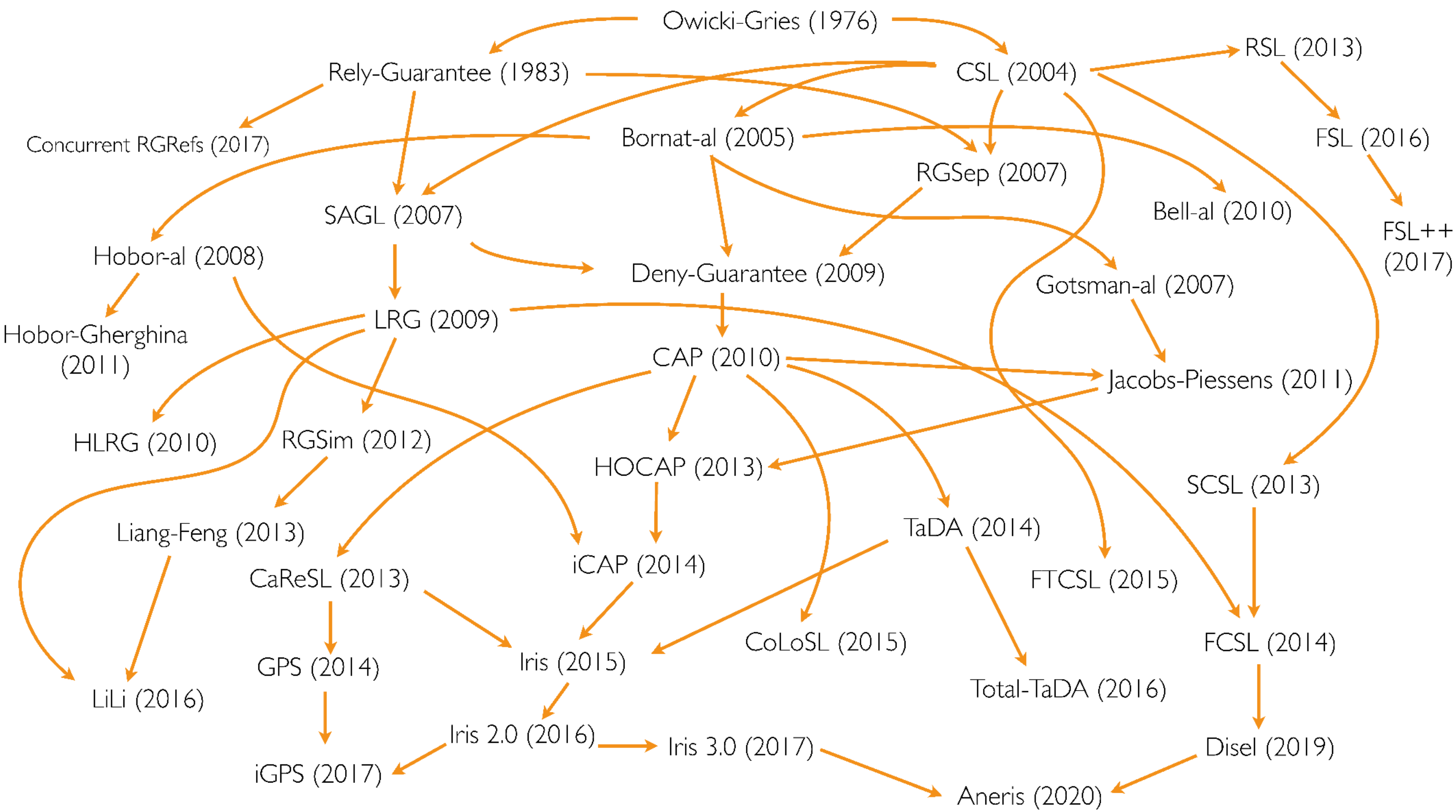
$\{\ x \mapsto (1,y) * y \mapsto (2,z)\ \}$ x.data=3; $\{\ x \mapsto (3,y) * y \mapsto (2,z)\ \}$



We can safely say that x.data is updated and y.data is still 2, because x cannot be aliased with y if the precondition is satisfied

Concurrent Separation Logics

CSL uses separating conjunction to do thread-local reasoning



But there are many flavors of CSL since O'Hearn's 2004 original

diagram: Ilya Sergey

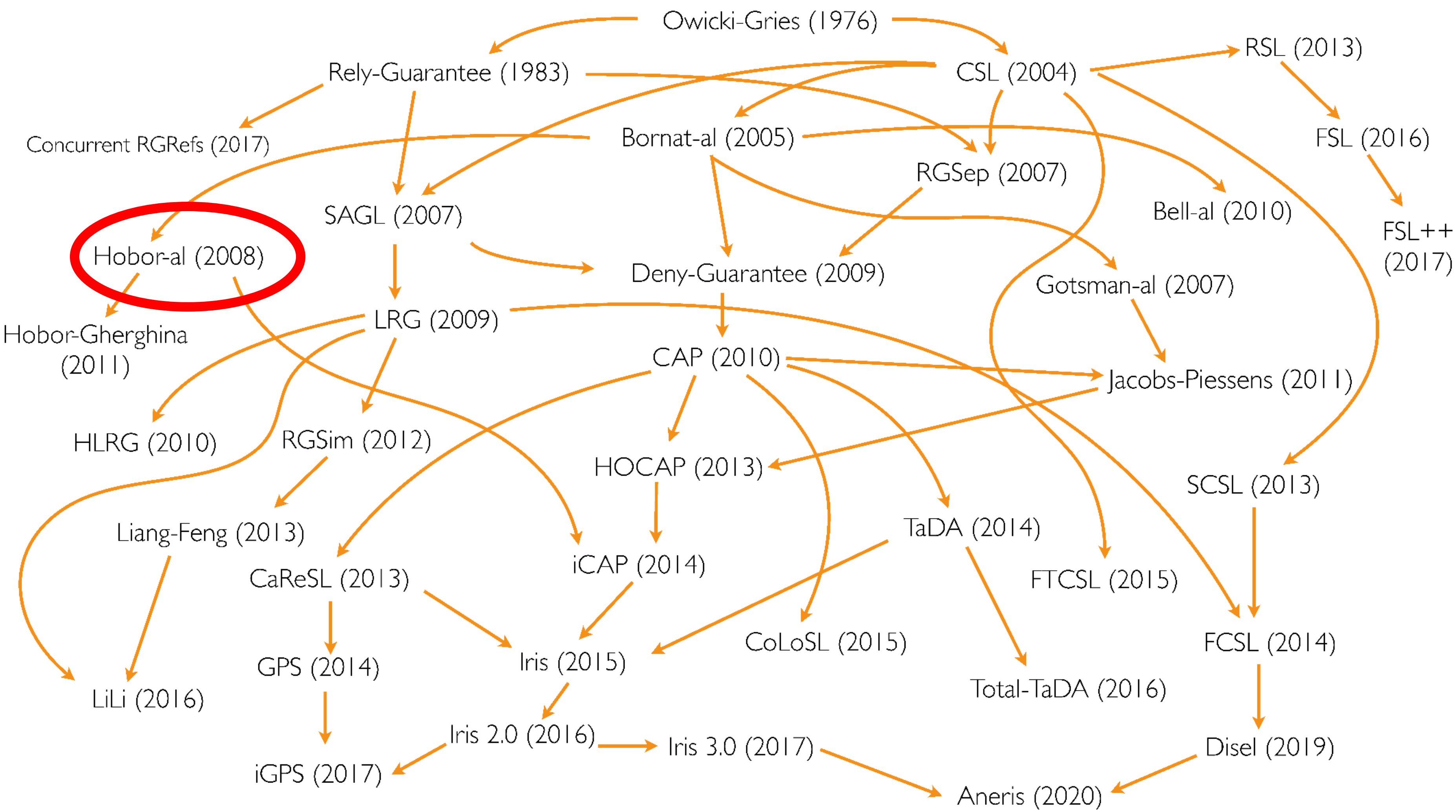
How to prove it

Some of those CSLs are quite complicated (but very expressive). But our needs here are simple:

- Don’t need ghost state
- Don’t need partial commutative monoid
- Semaphores with “old-fashioned” lock invariants
- Permission-splitting

∴ all the theory was in place by 2008!

Concurrent Separation Logics

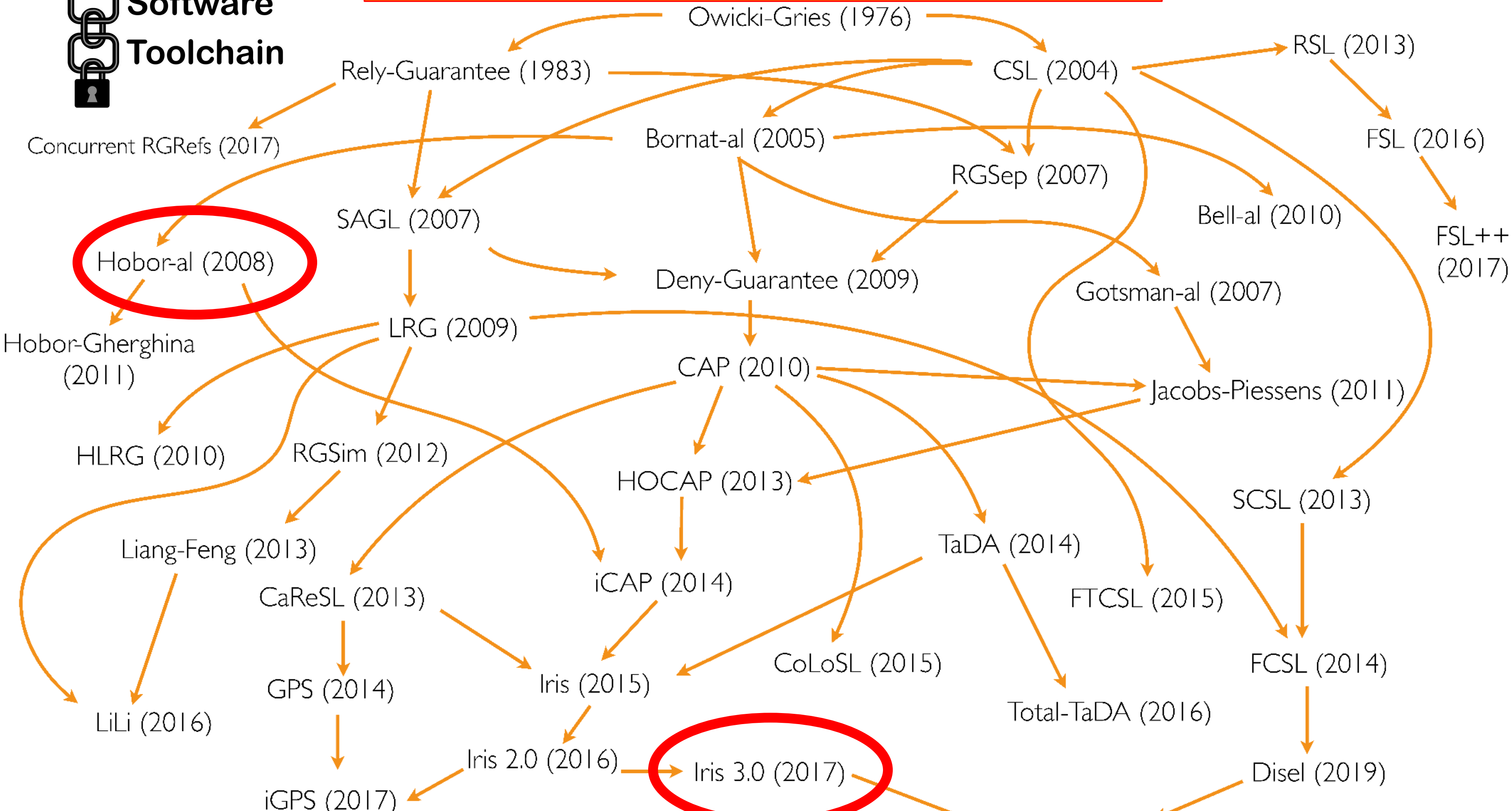


Oracle Semantics for Concurrent Separation Logic, by Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. *European Symposium on Programming (ESOP)*, 2008.

Concurrent Separation Logics

Verified
Software
Toolchain

What varieties does VST support, by the way?



Either Hobor-style or Iris-style, your choice

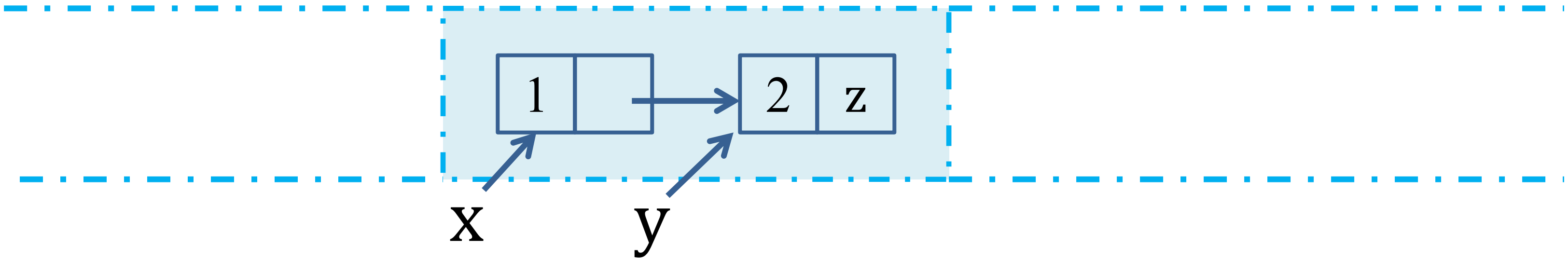
Resource invariants

- O'Hearn 2004
- Gotsman *et al.* 2007
- Hobor, Zappa Nardelli, Appel 2008

$$\{\, l \looparrowright R \,\}\ \text{acquire}(l)\ \{\overbrace{R}^{\text{acquire the lock, gain the resource}} * \ l \looparrowright R\}$$

$$\{\overbrace{R}^{\text{release the lock, give up the resource}} * \ l \looparrowright R\}\ \text{release}(l)\ \{l \looparrowright R\}$$

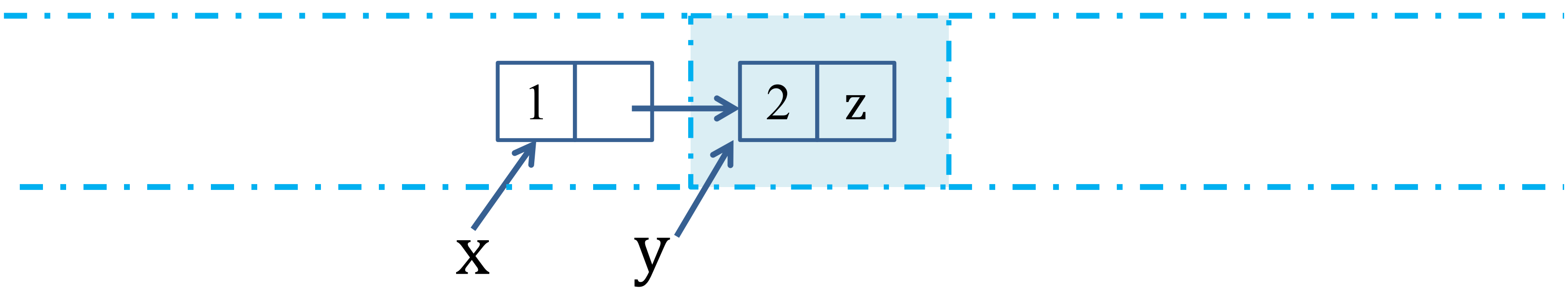
Heaplets in Separation Logic



$$\{\, x \mapsto (1,y) * y \mapsto (2,z) * l \looparrowright (\exists y.\ x \mapsto (1,y)) \}$$

$$\text{release}(l)$$

$$\{\, y \mapsto (2,z) * l \looparrowright (\exists y.\ x \mapsto (1,y)) \}$$



Heaplets in Separation Logic

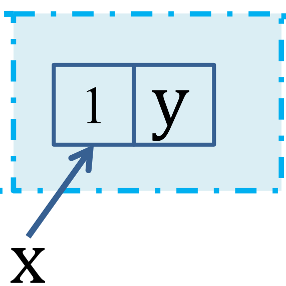
this is the resource invariant of the lock

$$\{l \hookrightarrow (\exists y.\ x \mapsto (1,y))\}$$

acquire the lock, gain the resource

$$\text{acquire}(l)$$

$$\{\ x \mapsto (1,y) * l \hookrightarrow (\exists y.\ x \mapsto (1,y))\}$$



Permission shares

- O'Hearn 2004
- Gotsman *et al.* 2007
- Hobor, Zappa Nardelli, Appel 2008

$$\{\mathrm{emp}\}\ l = \mathrm{makelock}(\,)\ \{l \leadsto R\}$$
$$\{\,l \leadsto R\}\ \mathrm{acquire}(l)\ \{R\ *\ l \leadsto R\}$$
$$\{R *\ l \leadsto R\}\ \mathrm{release}(l)\ \{l \leadsto R\}$$

split the permission share
into two parts

$$\frac{\pi = \pi_1 \oplus \pi_2}{p \mapsto_{\pi} v \quad \leftrightarrow \quad p \mapsto_{\pi_1} v\ *\ p \mapsto_{\pi_2} v}$$

split the "maps-to" resource
into two resources

Resource invariants for parsplit

```
void do_tasks(struct task *tasks, unsigned T) {
for (i=1; i<T; i++)
   release (tasks[i].go);
 tasks[0].f(tasks[0].closure);
 for (i=1; i<T; i++)
   acquire (tasks[i].done);
}
```

```
int thread_worker(void *arg) {
 struct task *t = (struct task *)arg;
 while (1)  {
   acquire(t->go);
   t->f(t->closure);
   release(t->done);
  }
}
```

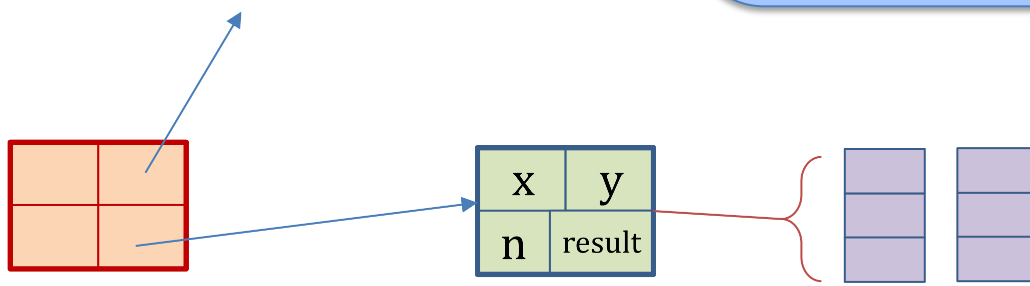
Release the go-lock,   thread_worker acquires it and starts working

Release the done-lock, task manager resumes collecting “done” statuses

Resource invariants for parsplit

```
void do_tasks(struct task *tasks, unsigned T) {
for (i=1; i<T; i++)
    release (tasks[i].go);
 tasks[0].f(tasks[0].closure);
 for (i=1; i<T; i++)
    acquire (tasks[i].done);
}
```

```
int thread_worker(void *arg) {
 struct task *t = (struct task *)arg;
 while (1)  {
    acquire(t->go);
    t->f(t->closure);
    release(t->done);
   }
}
```



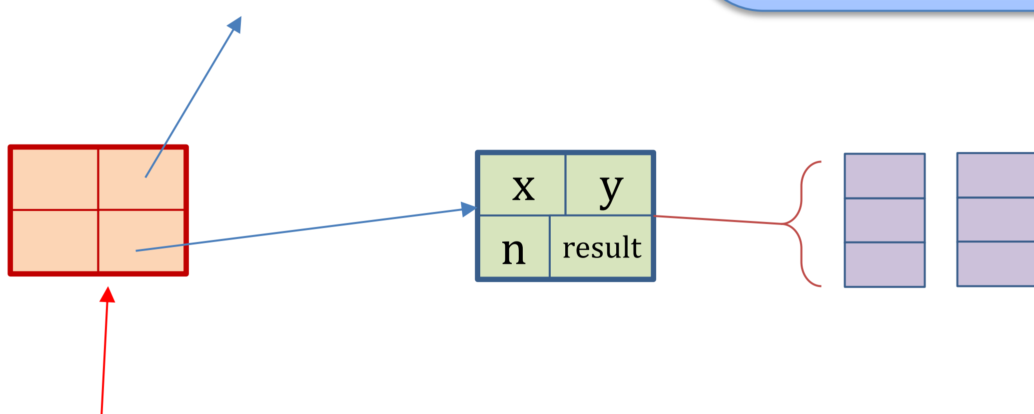
Definition task_inv $T\ q\ p \coloneqq \exists f \exists clo, (p.\mathrm{f}\ \mapsto_{\mathrm{r}} f) * (p.\mathrm{closure} \mapsto_{\mathrm{r}} clo) * \exists c, P(T, c, q, clo).$

We will use this definition in constructing resource invariants for **go** and **done** locks.

Resource invariants for parsplit

```
void do_tasks(struct task *tasks, unsigned T) {
for (i=1; i<T; i++)
    release (tasks[i].go);
 tasks[0].f(tasks[0].closure);
 for (i=1; i<T; i++)
    acquire (tasks[i].done);
}
```

```
int thread_worker(void *arg) {
 struct task *t = (struct task *)arg;
 while (1)  {
   acquire(t->go);
   t->f(t->closure);
   release(t->done);
  }
}
```



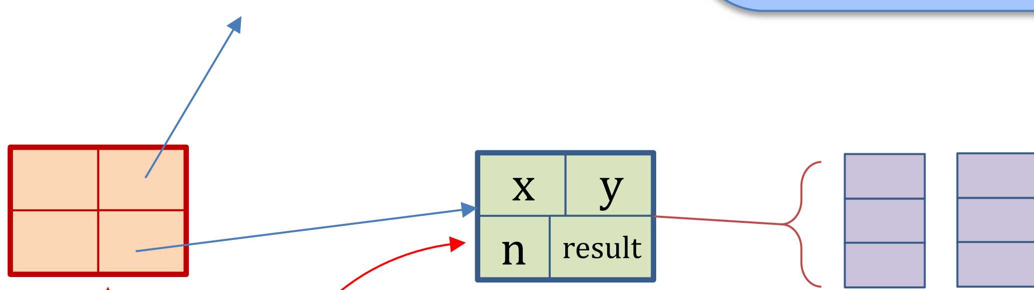
Definition task_inv $T\ q\ p := \exists f \exists clo, (p.\mathrm{f} \mapsto_{\mathrm{r}} f) * (p.\mathrm{closure} \mapsto_{\mathrm{r}} clo) * \exists c, P(T, c, q, clo).$

Argument p is the pointer to the task block (go,done,f,closure)

Resource invariants for parsplit

```
void do_tasks(struct task *tasks, unsigned T) {
for (i=1; i<T; i++)
    release (tasks[i].go);
 tasks[0].f(tasks[0].closure);
 for (i=1; i<T; i++)
    acquire (tasks[i].done);
}
```

```
int thread_worker(void *arg) {
 struct task *t = (struct task *)arg;
 while (1)  {
   acquire(t->go);
   t->f(t->closure);
   release(t->done);
  }
}
```



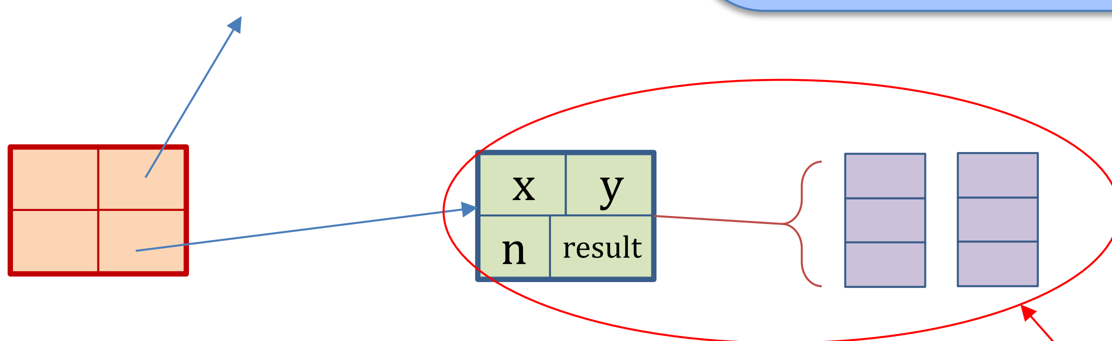
Definition task_inv $T\ q\ p := \exists f\, \exists clo, (p.\mathrm{f} \mapsto_{\mathrm{r}} f) * (p.\mathrm{closure} \mapsto_{\mathrm{r}} clo) * \exists c, P(T, c, q, clo).$

(existentially quantified) *clo* is the pointer to the dtask descriptor

Resource invariants for parsplit

```
void do_tasks(struct task *tasks, unsigned T) {
for (i=1; i<T; i++)
    release (tasks[i].go);
 tasks[0].f(tasks[0].closure);
 for (i=1; i<T; i++)
    acquire (tasks[i].done);
}
```

```
int thread_worker(void *arg) {
 struct task *t = (struct task *)arg;
 while (1)  {
   acquire(t->go);
   t->f(t->closure);
   release(t->done);
  }
}
```



Definition task_inv $T\ q\ p \coloneqq \exists f \exists clo, (p.\mathrm{f}\ \mapsto_{\mathrm{r}}\ f) * (p.\mathrm{closure} \mapsto_{\mathrm{r}}\ clo) * \exists c, P(T, c, q, clo).$

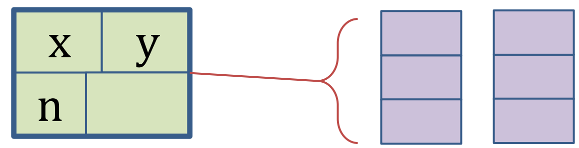
P is the (client-specific) task predicate (describing this slice of the x, y vectors)

Client-specific task predicate

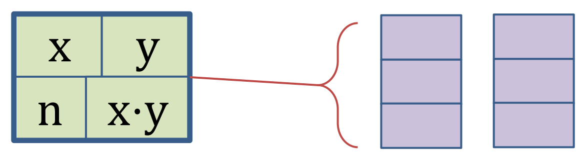
P is the client-specific (in this case, dot-product) predicate describing the state of the client task description

the ith `dtask` + the ith vector slices

$P(T, c, q, clo)$.



*This is the state when the **go** lock is released*



*This is the state when the **done** lock is released*

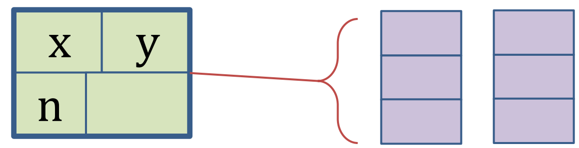
Definition task_inv $T\ q\ p \coloneqq \exists f \exists clo, (p.\mathrm{f} \mapsto_{\mathrm{r}} f) * (p.\mathrm{closure} \mapsto_{\mathrm{r}} clo) * \exists c, P(T, c, q, clo)$.

Questions and answers

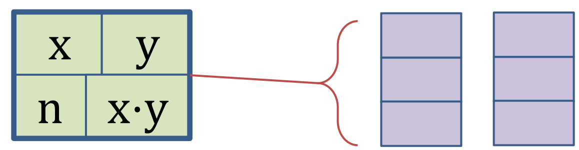
By releasing the **go** lock, we ask a question: what's the dot-product of this slice?
By releasing the **done** lock, worker thread answers the question.
The q parameter specifies whether we're asking or answering (go lock or done lock)

$P(T, c, q, clo)$.

q=ASK



q=ANSWER



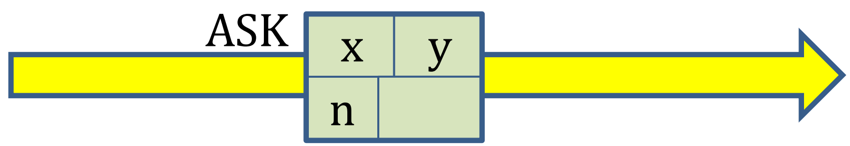
Definition task_inv $T\ q\ p \coloneqq \exists f \exists clo, (p.\mathrm{f} \mapsto_{\mathrm{r}} f) * (p.\mathrm{closure} \mapsto_{\mathrm{r}} clo) * \exists c, P(T, c, q, clo)$.

Resource invariants

<u>do_tasks</u>
```
for (i=1; i<T; i++)
    release (tasks[i].go);
for (i=1; i<T; i++)
    acquire (tasks[i].done);
}
```

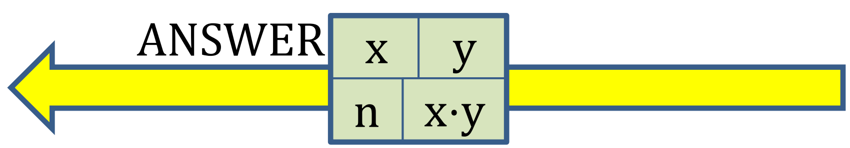
<u>thread_worker</u>
```
 while (1)  {
    acquire(t->go);
    t->f(t->closure);
    release(t->done);
  }
```

release(go)



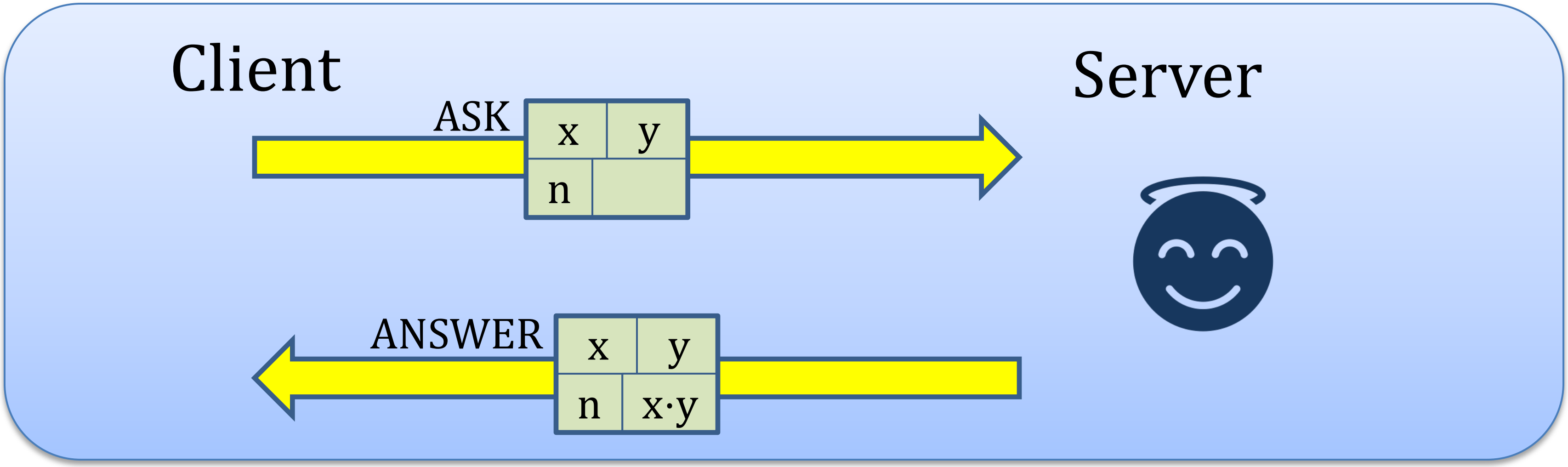
acquire(go)

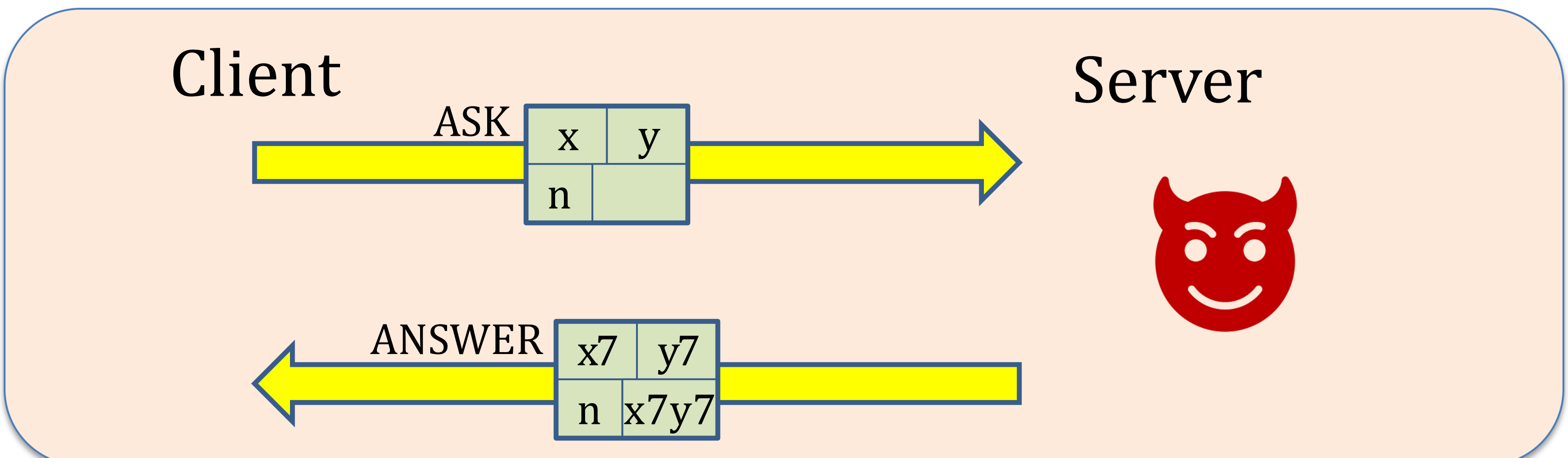
acquire(done)



release(done)

What question did I ask?

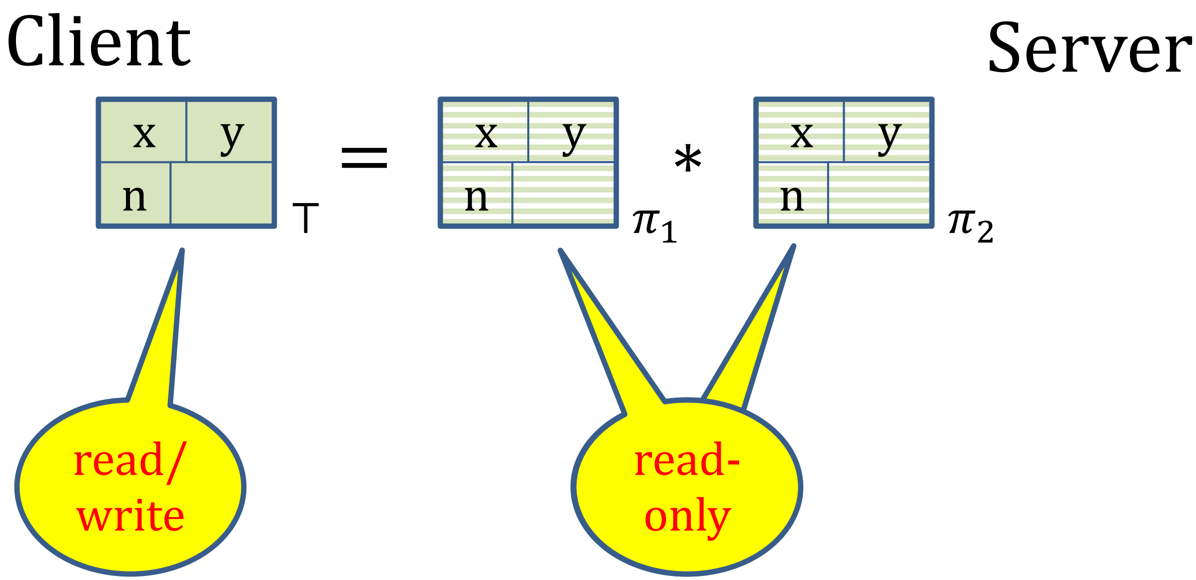




We must prevent the worker thread from changing the question, while permitting it to fill in the answer!

Splitting shares

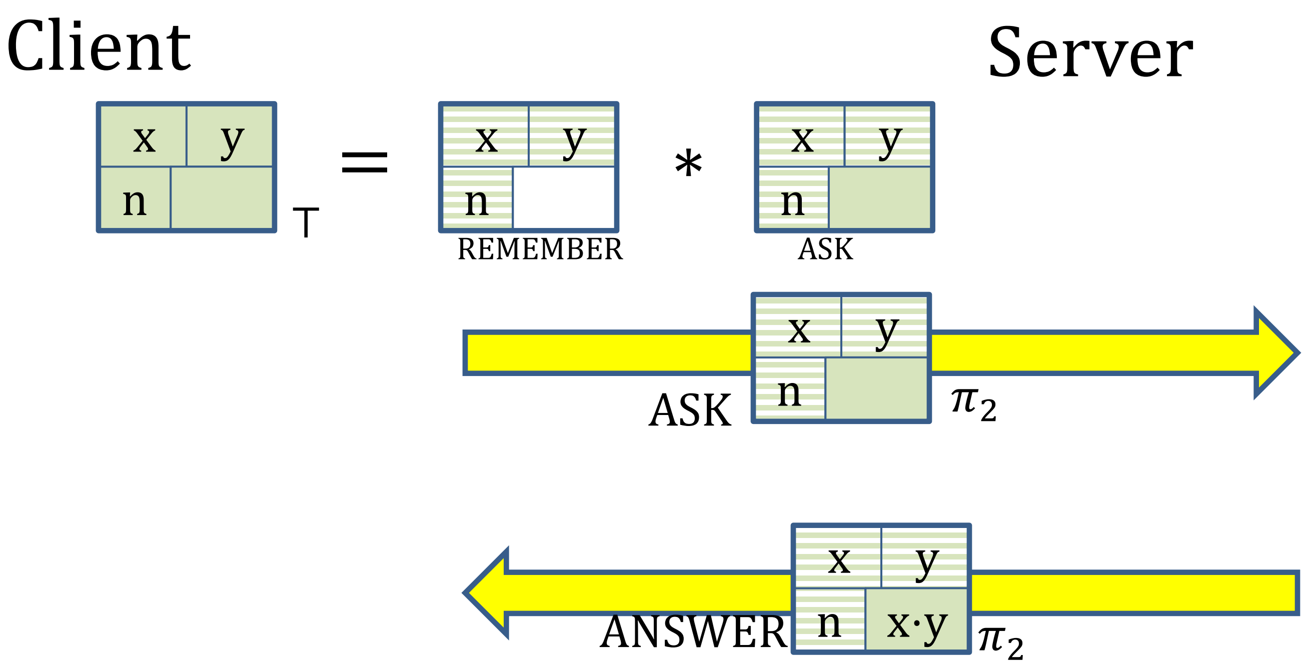
$$\frac{\pi = \pi_1 \oplus \pi_2}{p \mapsto_{\pi} v \quad \leftrightarrow \quad p \mapsto_{\pi_1} v \ * \ p \mapsto_{\pi_2} v}$$



A “writable” share may be split into two “readable” shares.

Splitting shares

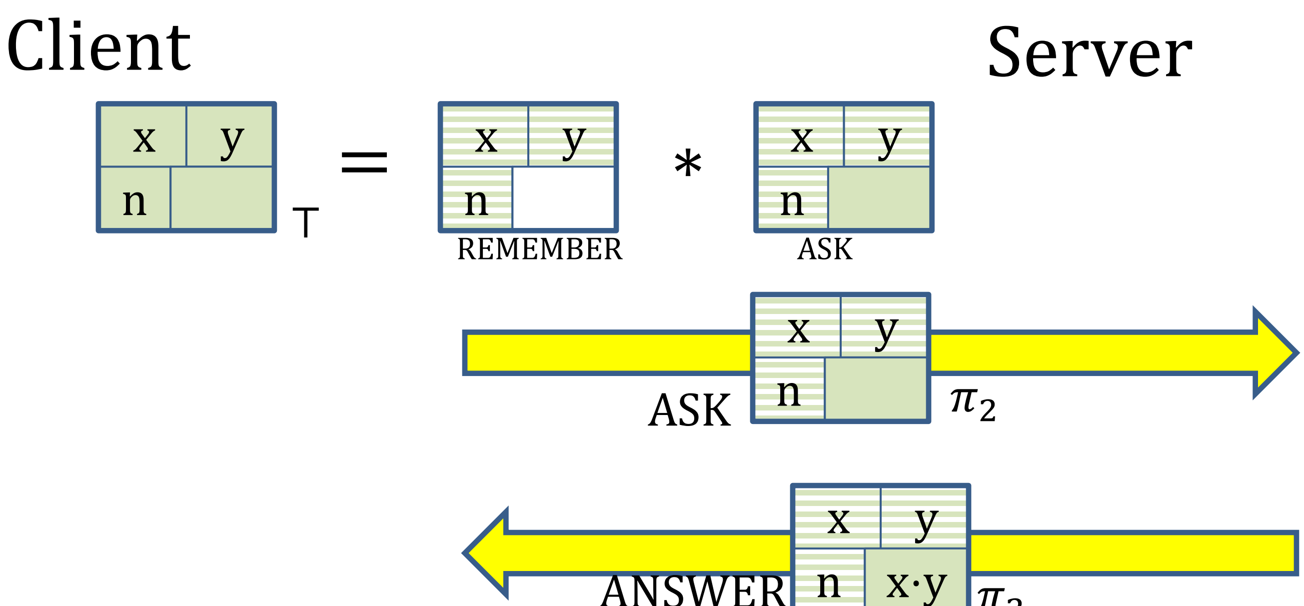
$$\frac{\pi = \pi_1 \oplus \pi_2}{p \mapsto_{\pi} v \quad \leftrightarrow \quad p \mapsto_{\pi_1} v \ * \ p \mapsto_{\pi_2} v}$$



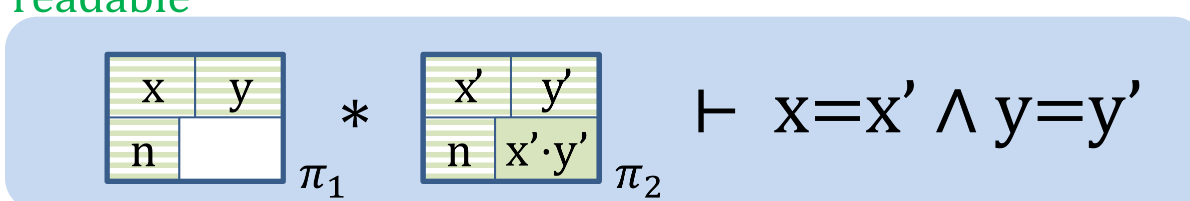
Split a "Top" share into a "Remember" share (readable x,y,n; no-access result) and an "Ask/Answer" share (readable x,y,n; writable result)

Splitting shares

$$\frac{\pi = \pi_1 \oplus \pi_2}{p \mapsto_{\pi} v \quad \leftrightarrow \quad p \mapsto_{\pi_1} v * p \mapsto_{\pi_2} v}$$

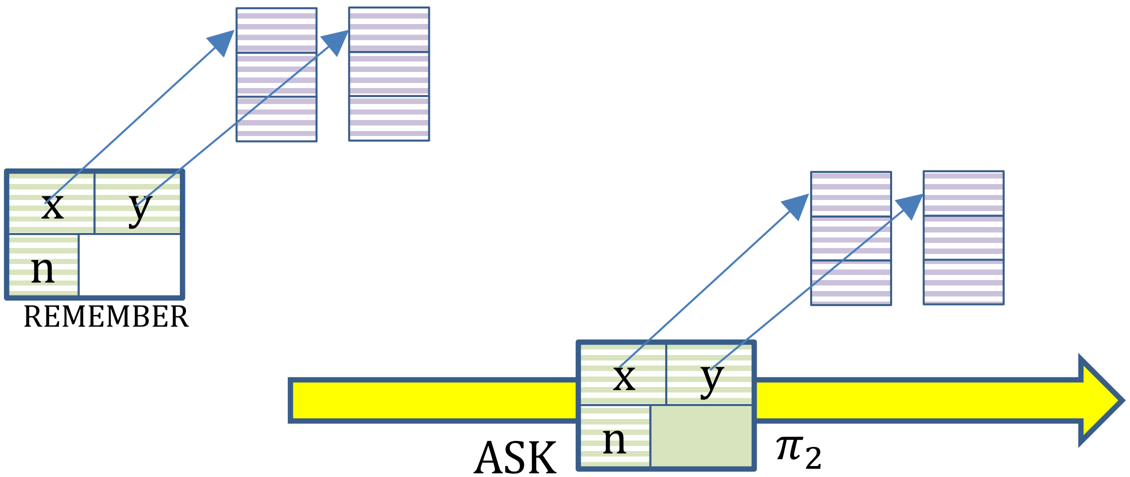


Fact: If you join two readable
shares, the values
must be the same!



So we make sure the answer corresponds to the <u>original</u> question.

Must also split shares of . . .



Not just the pointers x, y but also the pointed-to data
must be split into “Remember” share and “Ask/Answer” share

The functional model

Floating-point add is not associative, so cannot prove

$$\sum_{i=0}^{n} x_i \cdot y_i$$

Instead we prove

$$\sum_{t=0}^{T} \sum_{i=\delta_t}^{\delta_{t+1}} x_i \cdot y_i$$

and let the upper-layer proofs worry about accuracy of associativity

Q.E.D.

That’s my proof!

In Coq it’s a bit more verbose.

But does the program work?

```
$ time ./dotprod 1000000 4 10000
N=1000000  T=4  R=10000

real    10.415s
user    40.703s
sys      0.140s
```

speedup 3.9 with 4 processors

Lines of program and proof

	C lines	Coq lines	Ratio
dotprod client	58	799	14:1
API	10	148	15:1
parallelizer	51	493	10:1

Ugh!
Too much.

Where to find it

https://github.com/VeriNum/pardotprod

C program: parsplit.h, parsplit.c, dotprod.h, dotprod.c

Specifications: spec_parsplit.v, and dotprod_spec within verif_dotprod.v

Proofs: verif_parsplit.v, verif_dotprod.v

Conclusion

- Barrier synchronization is a simple parallel programming model, easy to implement with ordinary semaphores, quite useful in many parallel applications

- It's straightforward to specify and verify barrier-synch. parallelism in VST

- We have verified correctness of a simple task manager, useful for T-way fork-join parallelism.