Tâches

```
#pragma omp parallel // intialisé un sac de tâches pour l'équipe
{
…
#pragma omp task // initier une tâche, créer un sac pour ses sous tâches
{
      …
#omp taskwait // attendre la fin de toutes les tâches créées dans cette tâche
    ….
}

#pragma omp taskyield // passer la main à une autre tâche

…
#omp taskwait // attendre la fin de toutes les tâches créées dans le bloc
…
}
```

- Moyen de générer du parallélisme à grain fin
  - Surcoût moindre en comparaison à celui des threads
  - Simplifie parfois la programmation

Fibonacci

```
int fib ( int n )
{
If (n<2) return n;
return fib(n-1)+fib(n-2);
}
```

Fibonacci

```
int fib ( int n )
{
  int x,y;
  if ( n < 2 ) return n;
  #pragma omp task shared(x,n)
  x = fib(n-1);

  #pragma omp task shared(y,n)
  y = fib(n-2);

#pragma omp taskwait
return x+y;
}
```

Fibonacci

```
int fib ( int n )
{
  int x,y;
  if ( n < 2 ) return n;
  #pragma omp task shared(x,n)
  x = fib(n-1);

  #pragma omp task shared(y,n)
  y = fib(n-2);

#pragma omp taskwait
return x+y;
}
```

```
int main()
{
#pragma omp parallel
  #pragma omp single nowait
  result = comp_fib_numbers(10);
return EXIT_SUCCESS;
}
```

Optimisation ;-)

xeon 12 x 2

#pragma omp task shared(x,n) if (n > cut)

Ticks	#c	cut
18548	21	23
19597	15	21
19612	17	22
19658	21	21
20206	23	20
20386	17	20
20468	20	20
21220	22	21
…		
187324	24	22
190026	24	21
191123	24	21
194712	24	27
198414	24	23
214880	24	29
306759	24	28

If (n <= cut) fib_seq(n)

Ticks	#c	cut
2561	16	23
2627	17	23
2639	18	22
2666	18	23
2697	16	21
2738	19	23
…		
183286	24	21
183898	24	28
184823	24	23
185517	24	21
190627	24	27
195394	24	24

cut	#tâches
21	176
22	108
23	66
24	40
25	24
26	14

Séquentiel 12057

```
(for j in $(seq 15 29); do  for i in $(seq 24); do OMP_NUM_THREADS=$i ./fibo 30 $j ; done ; done) > out
```

Optimisation ;-)

opteron 48 cœurs

#pragma omp task shared(x,n) if (n > cut)

Ticks	#c	cut
12433	37	22
12717	31	24
12834	33	24
12994	35	24
13075	34	24
13115	32	24
13206	44	22
13221	37	24
13308	43	22
…		
162846	1	17
163027	1	22
163029	1	21
163459	1	18
164070	1	28
164882	1	29

If (n <= cut) fib_seq(n)

Ticks	#c	cut
3059	11	26
3078	15	24
3262	13	24
3265	13	26
3279	10	24
3298	20	25
3312	18	24
3320	17	24
…		
128456	44	15
132008	47	15
140872	48	15
141058	45	15
149382	46	15

cut	#tâches
21	176
22	108
23	66
24	40
25	24
26	14

Séquentiel 14318

```
(for j in $(seq 15 29); do  for i in $(seq 48); do OMP_NUM_THREADS=$i ./fibo 30 $j ; done ; done) > out
```

Tâches

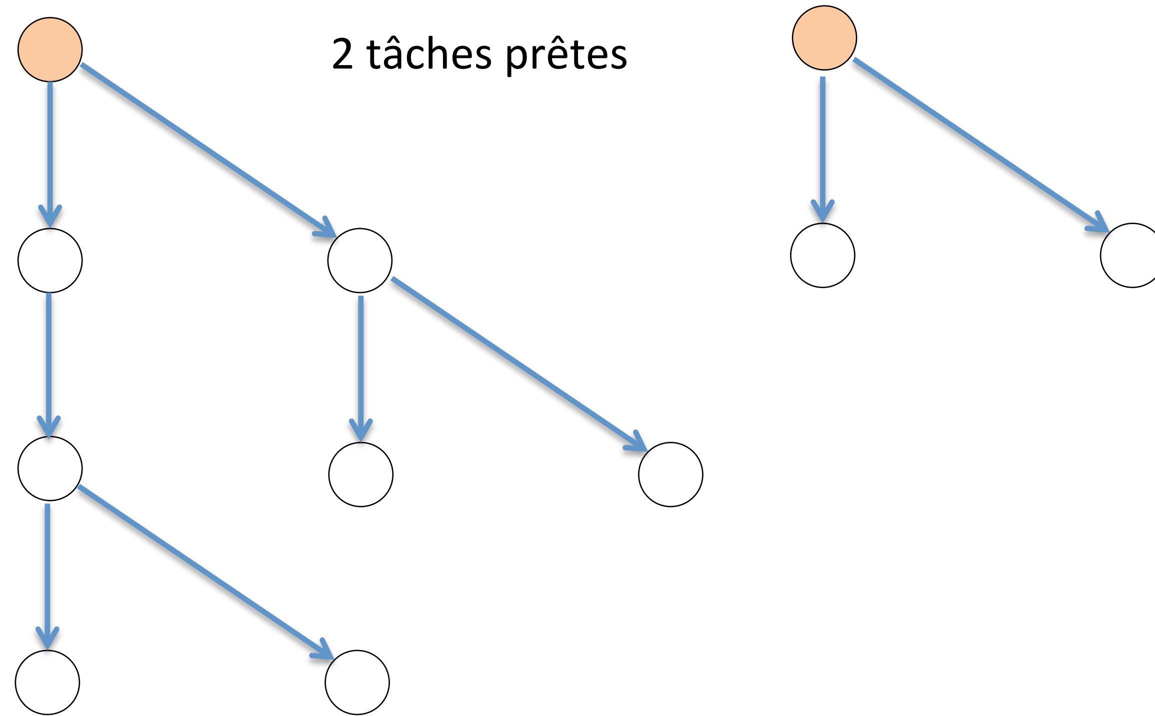
Récapitulatif

- Pas de miracle
  - Nécessaire maîtrise de la granularité
- Clauses
  - if(*scalar-logical-expression*)
  - untied
  - default(private | firstprivate | shared | none)
  - private(*list*) firstprivate(*list*) shared(*list*)
  - OMP 3.1 : optimisation de la création des tâches
    - Final(*scalar-logical-expression*) : fin du //isme
    - Mergeable : ne pas créer d'environnement

Ordonnancement des tâches

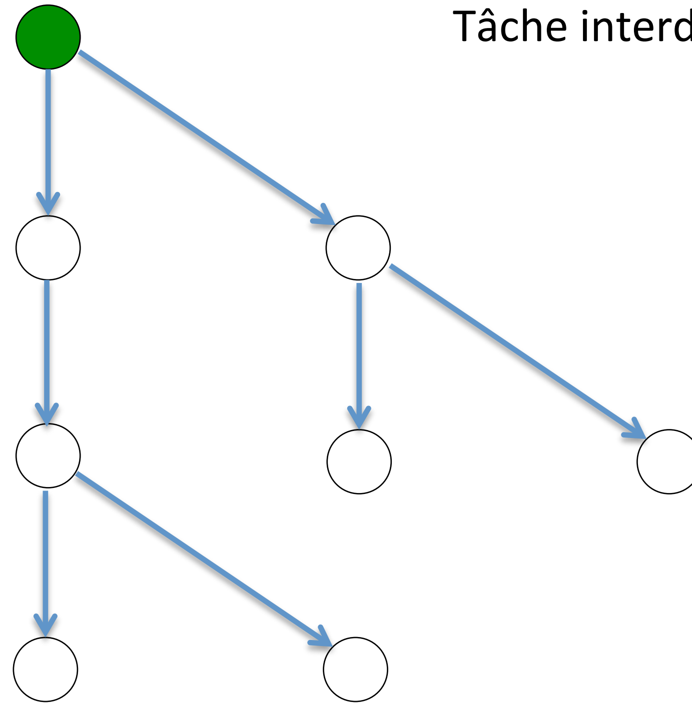
- Une tâche initiée peut être exécutée par tout thread de l'équipe du thread initiateur.
- Un thread ayant pris en charge l'exécution d'une tâche doit la terminer :
  - à moins que la clause *untied* ait été spécifiée,
  - mais peut l'exécuter en plusieurs fois.
- Un thread peut prendre / reprendre en charge une tâche :
  - à la création d'une tâche
  - à la terminaison d'une tâche
  - à la rencontre d'un taskwait, d'une barrière implicite ou non
- Quelle tâche ?
  - *In order to start the execution of a new tied task, the new task must be a descendant of every suspended task tied to the same thread, unless the encountered task scheduling point corresponds to a barrier region.*

Quelle tâche ?

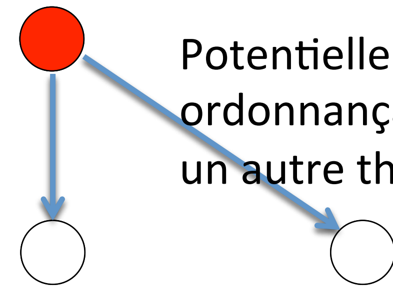


Quelle tâche ?

On exécute la tâche



Tâche interdite

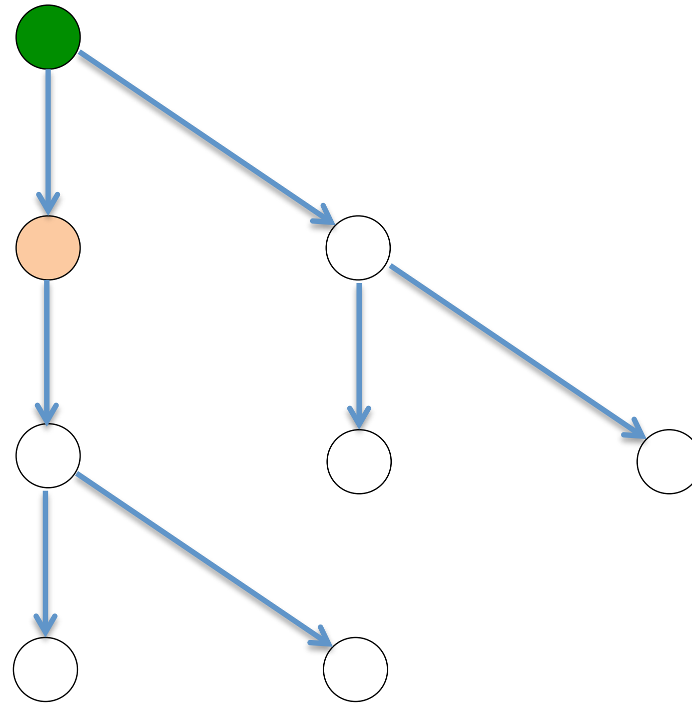


Potentiellement ordonnançable par un autre thread

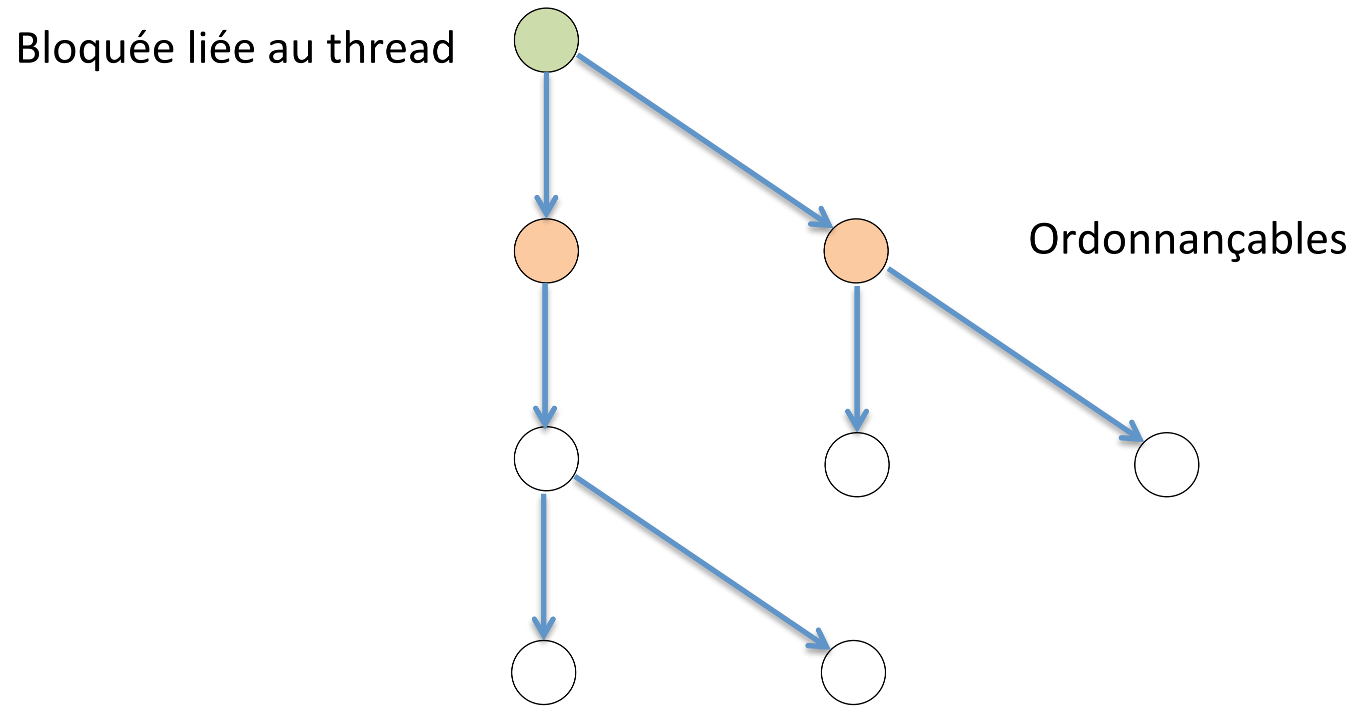
Quelle tâche ?

On exécute la tâche

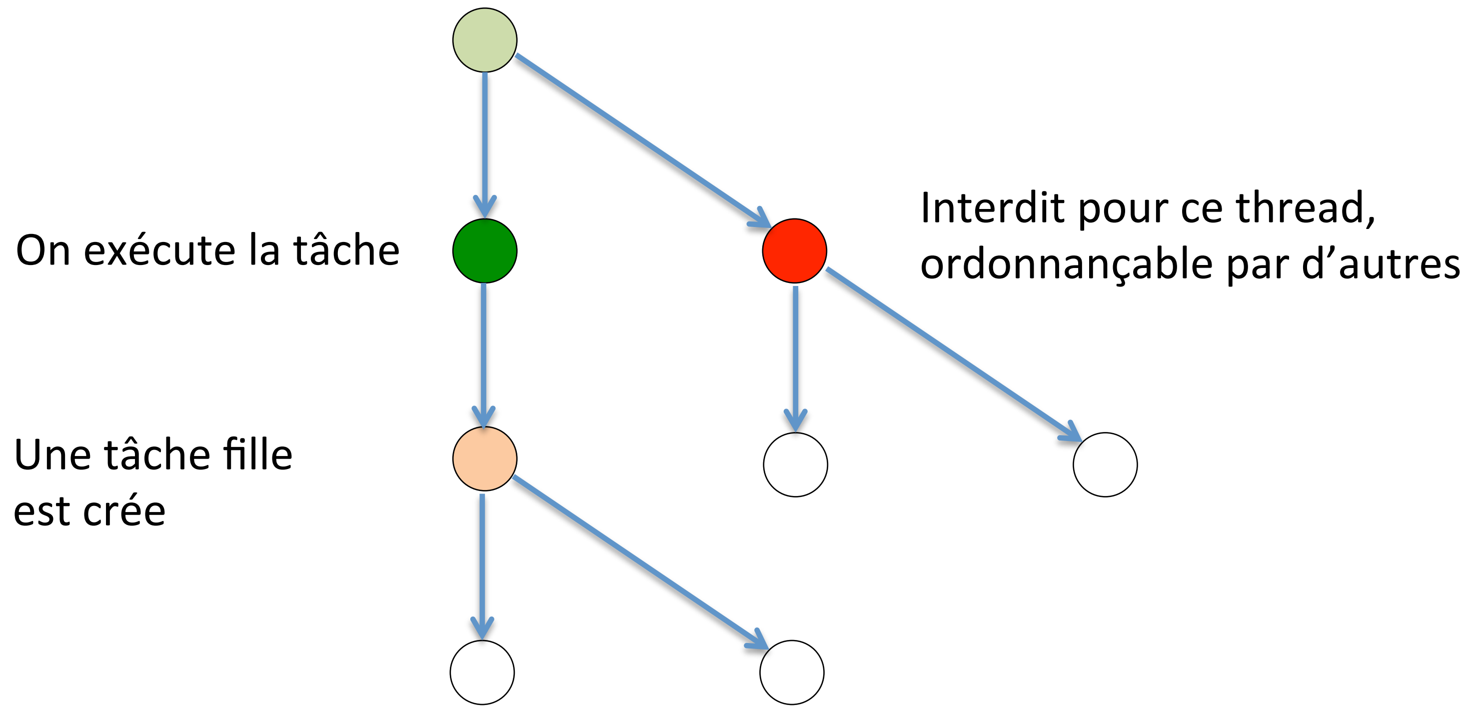
Tâche fille
potentiellement
ordonnançable
par tout thread
de l'équipe



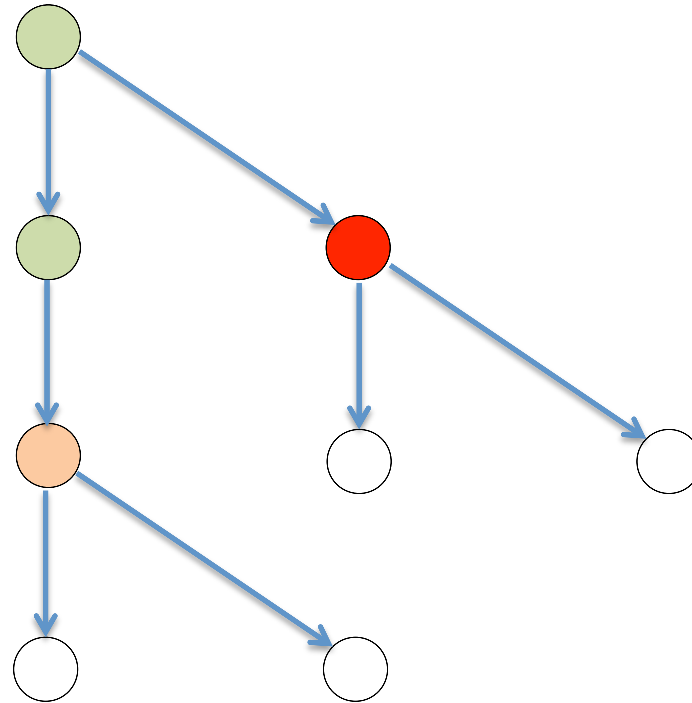
Quelle tâche ?



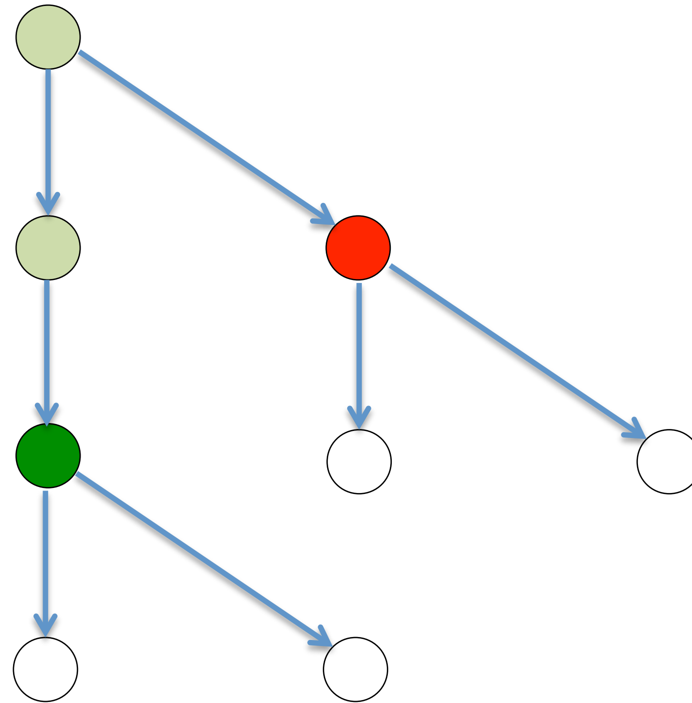
Quelle tâche ?



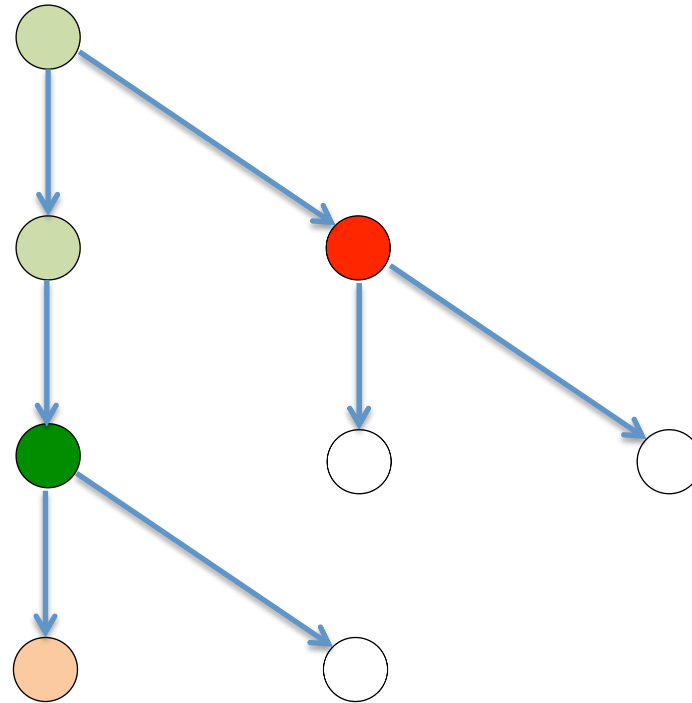
Quelle tâche ?



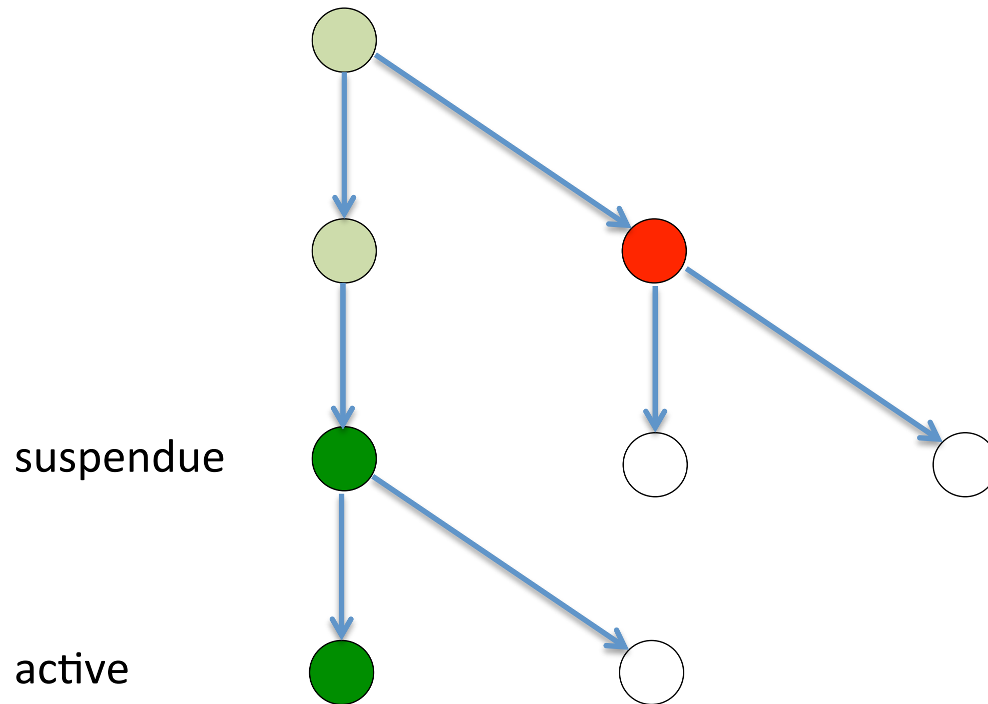
Quelle tâche ?



Quelle tâche ?

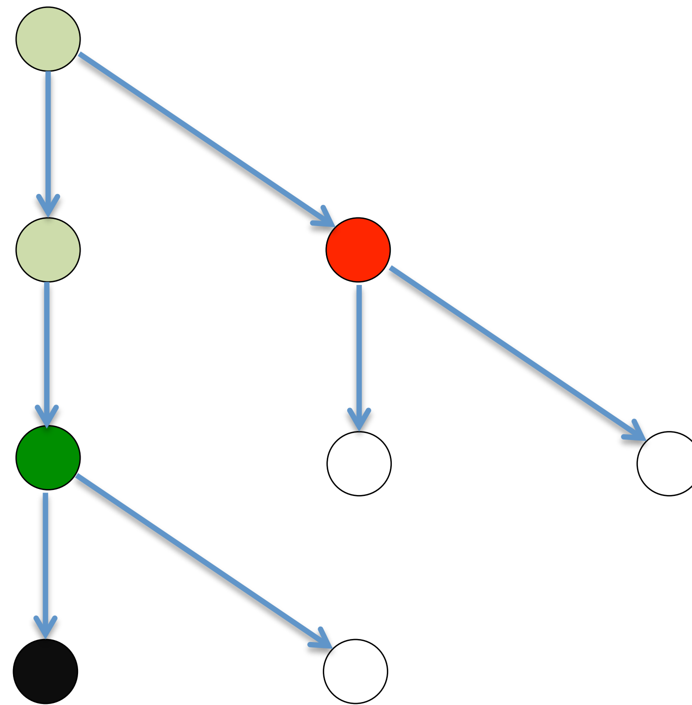


Quelle tâche ?

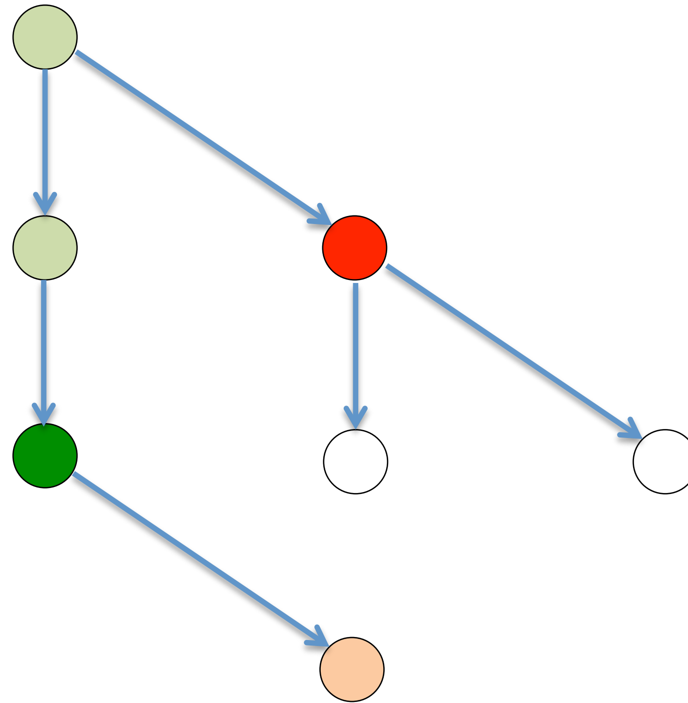


2 tâches
ordonnançables

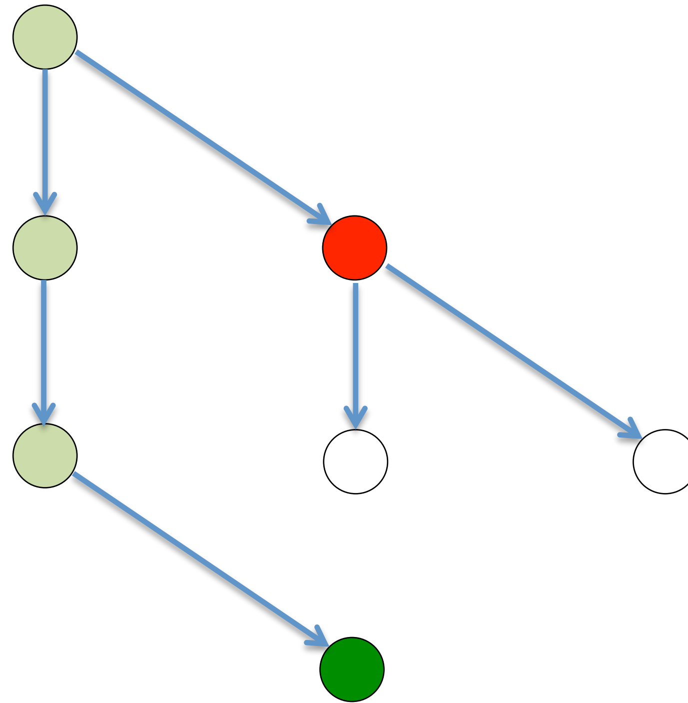
Quelle tâche ?



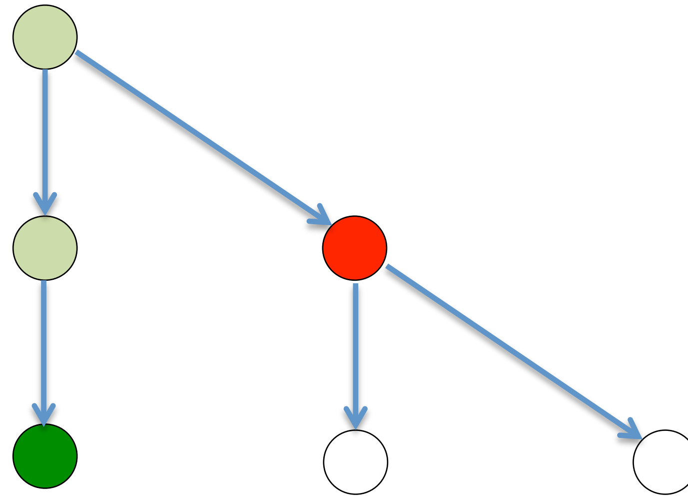
Quelle tâche ?



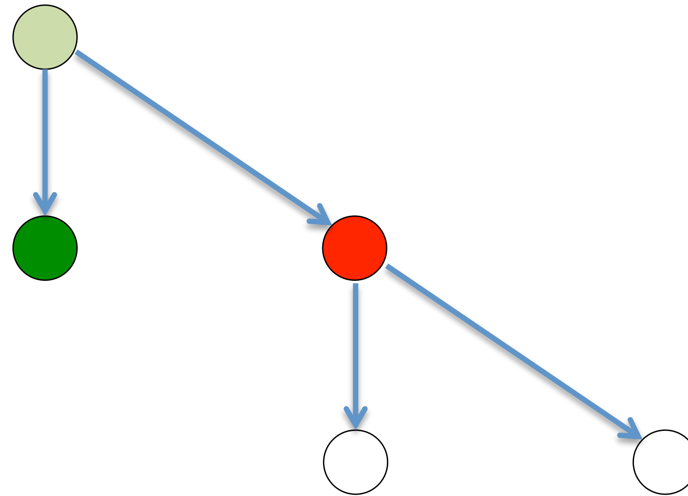
Quelle tâche ?



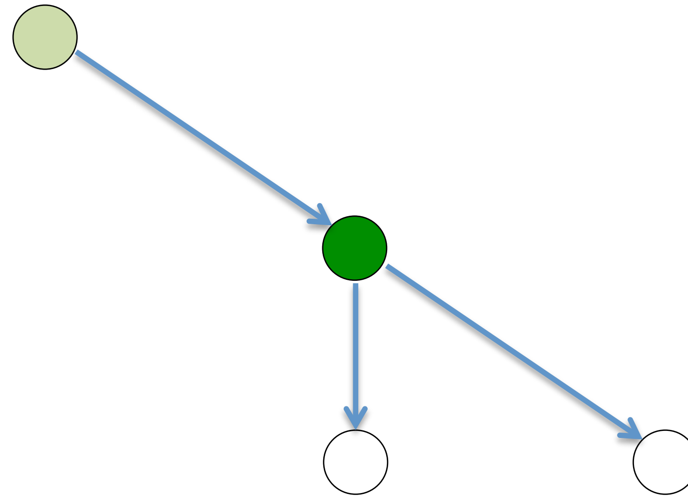
Quelle tâche ?



Quelle tâche ?



Quelle tâche ?



Ordonnancement des tâches

untied

- Autorise une tâche suspendue à être exécutée (i.e. reprise) par tout thread de l'équipe
  - Elle peut être suspendue à tout moment.
- Eviter *untied* en même temps :
  - Variables *threadprivate*
  - Identité du thread (omp_thread_number)
  - Critical et locks
- Astuce (?)
  - #pragma omp task if(0)

Quelques exemples

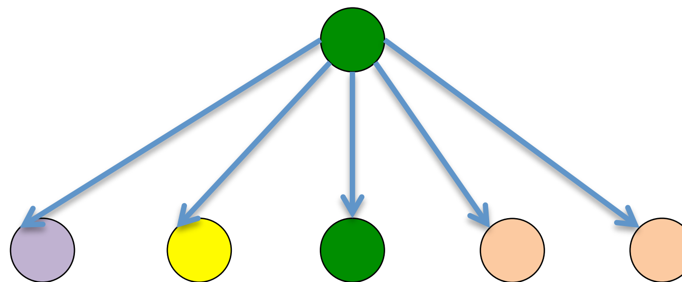
- Traversée de liste
- Quick sort
- Réduction à l'aide de tâche
- Dépendance entre tâches

Traversée de liste

```
#pragma omp parallel
#pragma omp single private(e)
{
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```

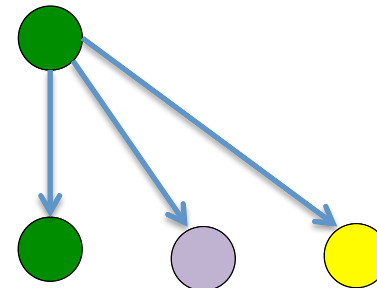
Traversée de liste

```
#pragma omp parallel
#pragma omp single private(e)
{
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```



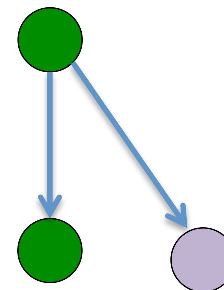
Traversée de liste

```
#pragma omp parallel
#pragma omp single private(e)
{
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```



Traversée de liste

```
#pragma omp parallel
#pragma omp single private(e)
{
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```



Traversée de liste

```
#pragma omp parallel
#pragma omp single private(e)
{
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```

Traversée de liste

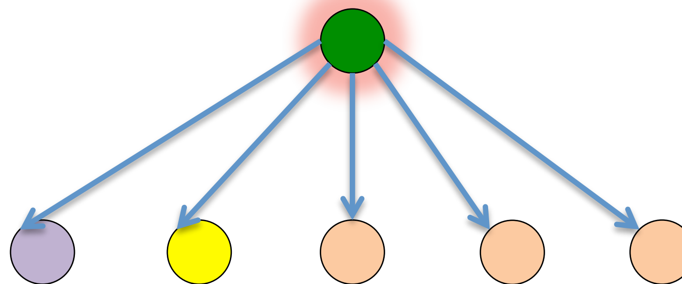
ne pas bloquer la tâche génératrice

```
#pragma omp parallel
#pragma omp single private(e)
{
#pragma omp task untied
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```

Traversée de liste

ne pas bloquer la tâche génératrice

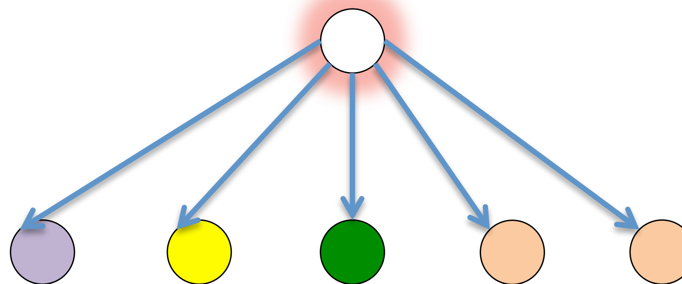
```
#pragma omp parallel
#pragma omp single private(e)
{
#pragma omp task untied
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```



Traversée de liste

ne pas bloquer la tâche génératrice

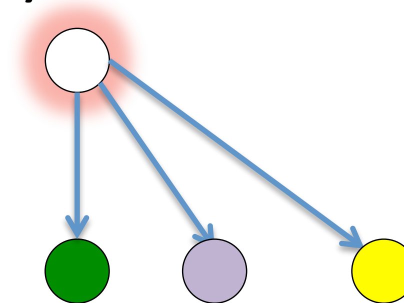
```
#pragma omp parallel
#pragma omp single private(e)
{
#pragma omp task untied
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```



Traversée de liste

ne pas bloquer la tâche génératrice

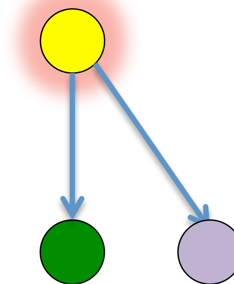
```
#pragma omp parallel
#pragma omp single private(e)
{
#pragma omp task untied
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```



Traversée de liste

ne pas bloquer la tâche génératrice

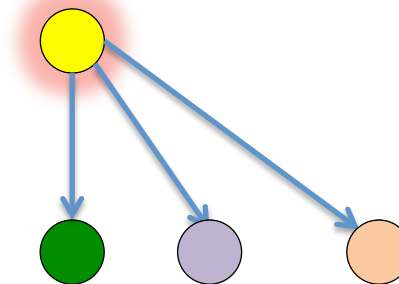
```
#pragma omp parallel
#pragma omp single private(e)
{
#pragma omp task untied
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```



Traversée de liste

ne pas bloquer la tâche génératrice

```
#pragma omp parallel
#pragma omp single private(e)
{
#pragma omp task untied
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```



Qsort

http://wikis.sun.com/display/openmp

```
void quick_sort (int p, int r, float *data)
{
  if (p < r) {
      int q = partition (p, r, data);
      #pragma omp parallel sections firstprivate(data, p, q, r)
      {
        #pragma omp section
        quick_sort (p, q-1, data, low_limit);
        #pragma omp section
        quick_sort (q+1, r, data, low_limit);
      }
    }
  }
}
```

Qsort

http://wikis.sun.com/display/openmp

```
void quick_sort (int p, int r, float *data)
{
   if (p < r) {
       int q = partition (p, r, data);
        #pragma omp task

        quick_sort (p, q-1, data, low_limit);
        #pragma omp task
        quick_sort (q+1, r, data, low_limit); }
      }
   }
}
```

```
void par_quick_sort (int n, float *data)
{
   #pragma omp parallel
   {
      #pragma omp single nowait
      quick_sort (0, n, data);
   }
```

Comparaison par Sun tâche vs section

*First, we set environment variable OMP_THREAD_LIMIT so that the program will use a **max of 8 threads**. The experiment was performed on a machine **with 8 processors**. The number of threads for the parallel region example is set by setting the OMP_NUM_THREADS environment variable. We obtain these performance numbers:*

OMP_NUM_THREADS	**task**	**parreg**
2	2.6s	1.8s
4	1.7s	2.1s
8	1.2s	2.6s

- *The program written with tasks achieves a good scalable speedup, while the program written with nested parallel regions does not.*
- Les threads s'attendent à la fin des sections
  - Seulement deux threads travaillent lorsque OMP_NUM_THREADS = 8 !

Tâches & Réduction atomic

```
int count_good (item_t *item)
{
  int n = 0;
  #pragma omp parallel
    #pragma omp single nowait
    {
        while (item) {
            #pragma omp task firstprivate(item)
               if (is_good(item)) {
                  #pragma omp atomic
                  n ++;
               }
            item = item->next;
        }
  return n;
}
```

Tâches & Réduction thread_num

```
int n = 0;
int pn[P]; // P is the number of threads used.
#pragma omp parallel
{
  pn[omp_get_thread_num()] = 0;
  #pragma omp single nowait
  {
     while (item) {
         #pragma omp task firstprivate(item)
            if (is_good(item)) {
               pn[omp_get_thread_num()] ++;
            }
         item = item->next;
     }
  }
  #pragma omp atomic
  n += pn[omp_get_thread_num()];
}
```

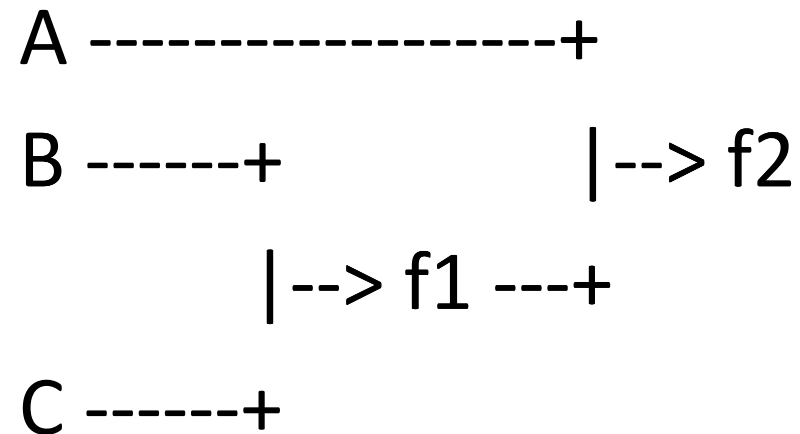
Tâches & Réduction atomic

```
int n = 0;
int pn[P]; // P is the number of threads used.
#pragma omp parallel
{
  pn[omp_get_thread_num()] = 0;
  #pragma omp single nowait
  {
     while (item) {
         #pragma omp task firstprivate(item)
            if (is_good(item)) {
              pn[omp_get_thread_num()] ++;
            }
         item = item->next;
     }
  }
  #pragma omp atomic
  n += pn[omp_get_thread_num()];
}
```

Dépendance de tâches
Bricolage

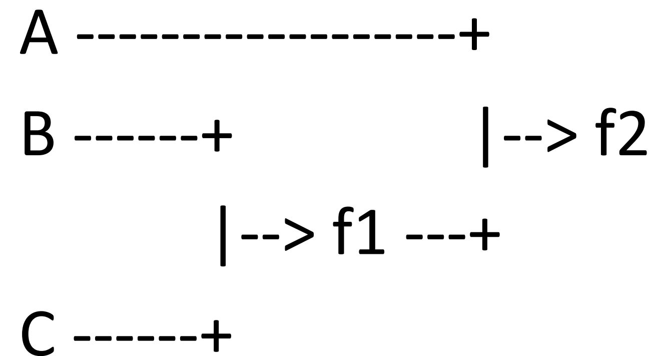
```
int foo ()
{
    int a, b, c, x, y;
    a = A();
    b = B();
    c = C();
    x = f1(b, c);
    y = f2(a, x);
     return y;
}
```

Avec les dépendances



Dépendance de tâches
Bricolage

```
#pragma omp task
 a = A();
 #pragma omp task if (0)
 {
    #pragma omp task
    b = B();
    #pragma omp task
    c = C();
    #pragma omp taskwait
 }
 x = f1 (b, c);
 #pragma omp taskwait
 y = f2 (a, x);
```



Autres modèles
Cilk

- Cilk MIT (Leiserson) depuis 1995
  - Basé sur le vol de travail et les continuations
  - Approche « diviser pour régner »
  - Structuration du parallélisme
  - Cadre théorique pour apprécier les performances
  - Intel Cilk plus

- TBB = cilk + OpenMP + c++

Autres modèles
Cilk

- Cilk MIT (Leiserson) depuis 1995
  - Basé sur le vol de travail et les continuations
  - Approche « diviser pour régner »
  - Structuration du parallélisme
  - Cadre théorique pour apprécier les performances

Basic Cilk Keywords

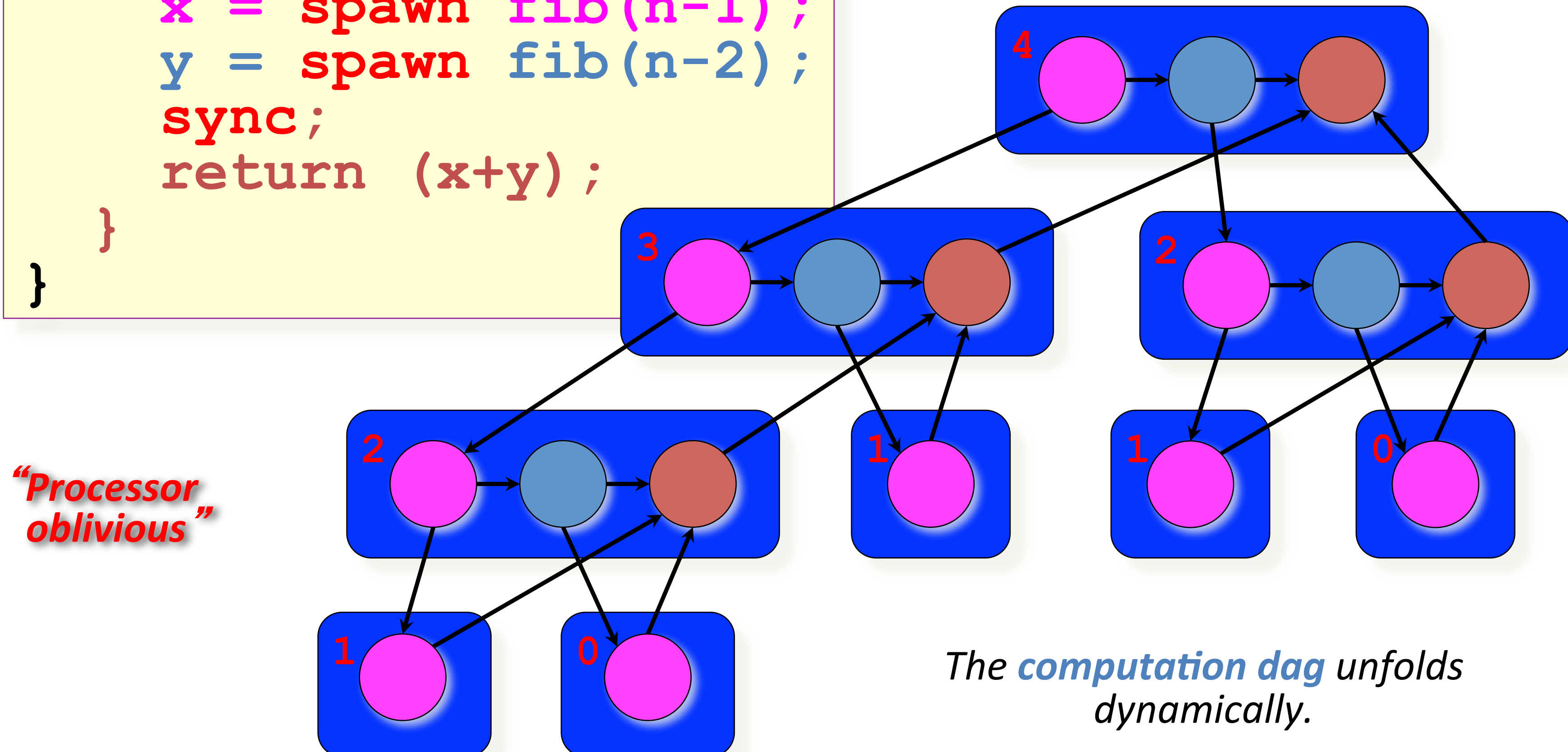
```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

Identifies a function as a ***Cilk procedure***, capable of being spawned in parallel.

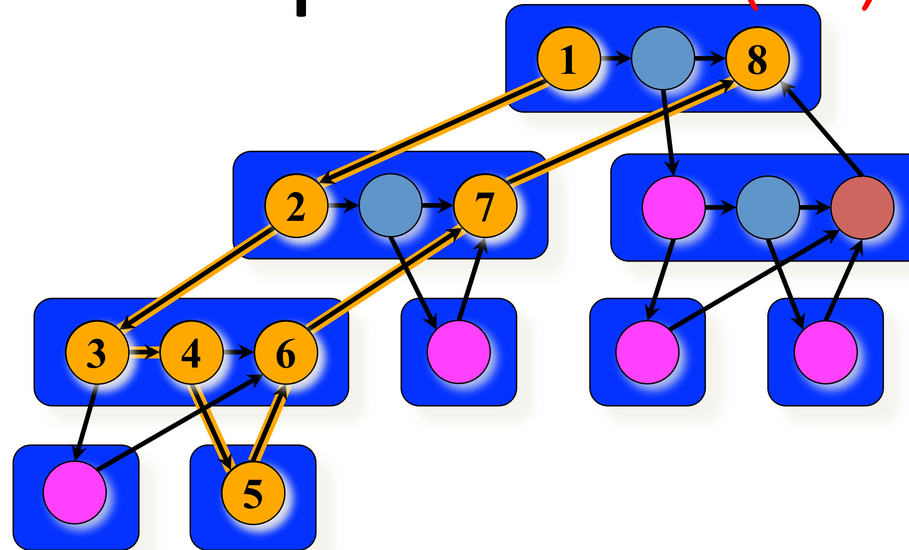
The named ***child*** Cilk procedure can execute in parallel with the ***parent*** caller.

Control cannot pass this point until all spawned children have returned.

Dynamic Multithreading



Example: fib(4)

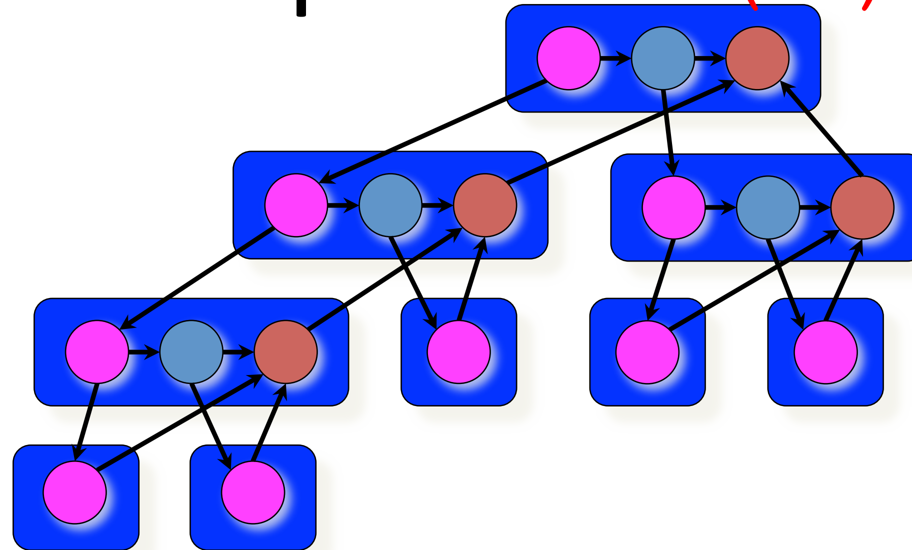


Assume for simplicity that each Cilk thread in **fib()** *takes unit time to execute.*

Work: $T_1 = 17$

Span: $T_\infty = 8$

Example: `fib(4)`



Assume for simplicity that each Cilk thread in **`fib()`** *takes unit time to execute.*

Work: $T_1 = 17$

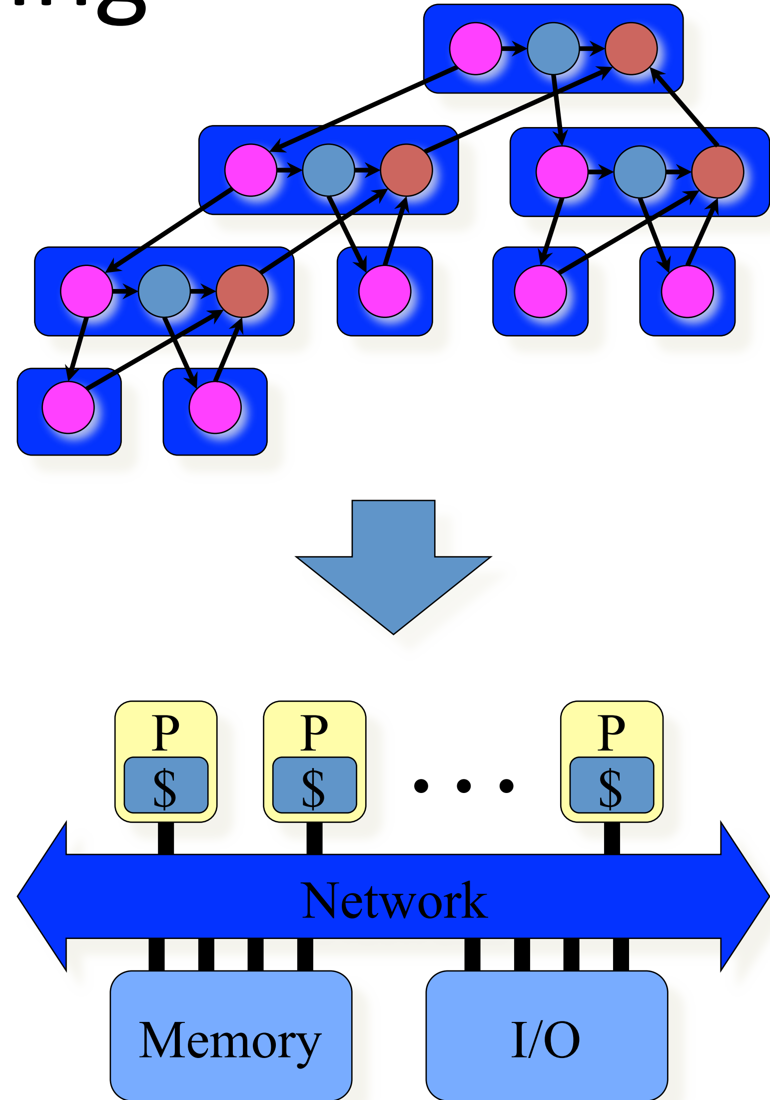
Span: $T_\infty = 8$

Parallelism: $T_1/T_\infty = 2.125$

Using many more than 2 processors makes little sense.

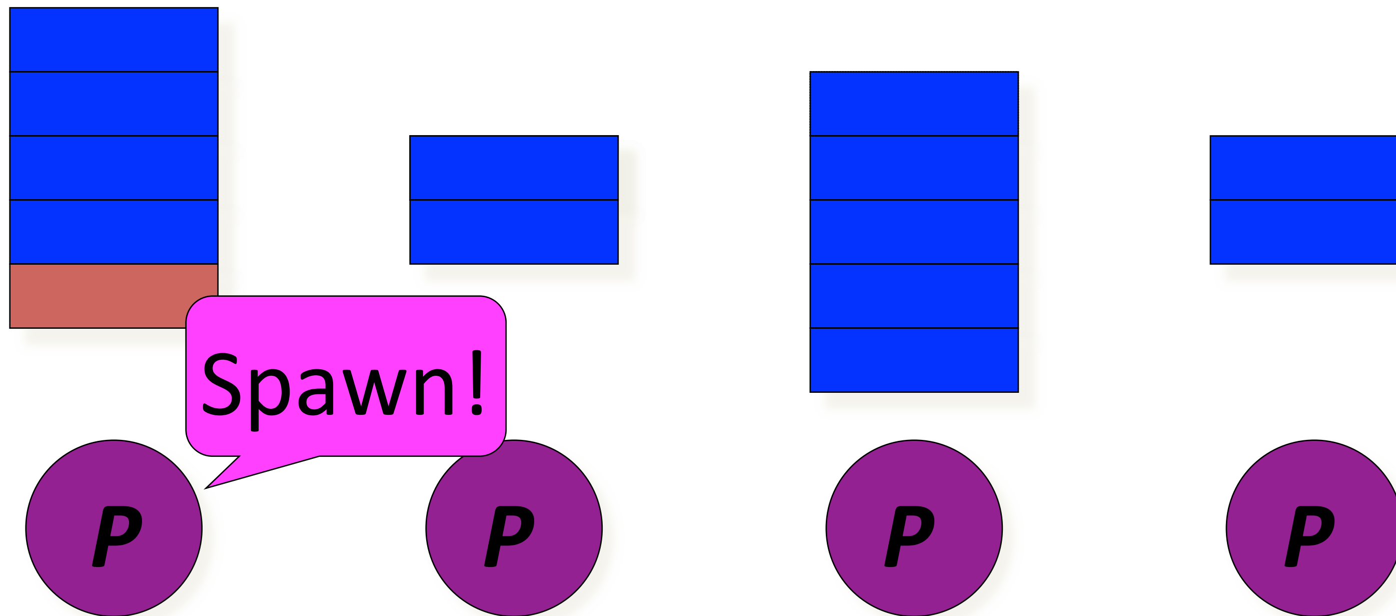
Scheduling

- Cilk allows the programmer to express *potential* parallelism in an application.
- The Cilk ***scheduler*** maps Cilk threads onto processors dynamically at runtime.
- Since ***on-line*** schedulers are complicated, we'll illustrate the ideas with an ***off-line*** scheduler.



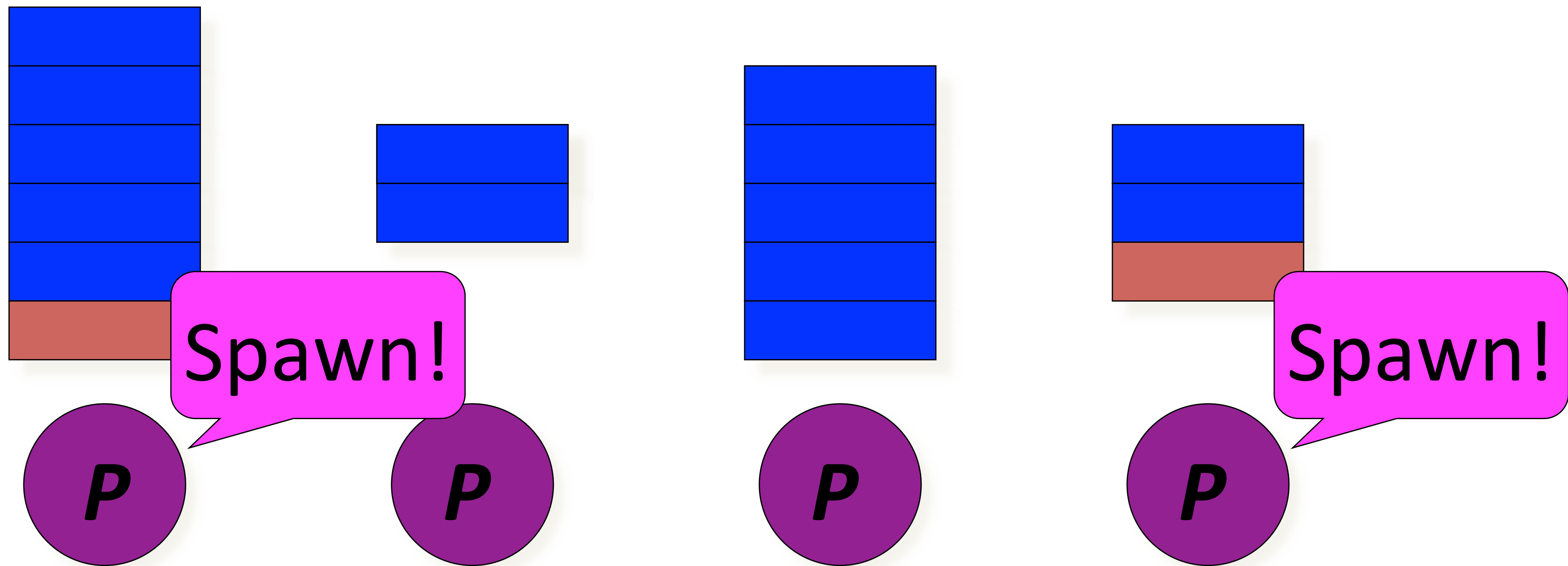
Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



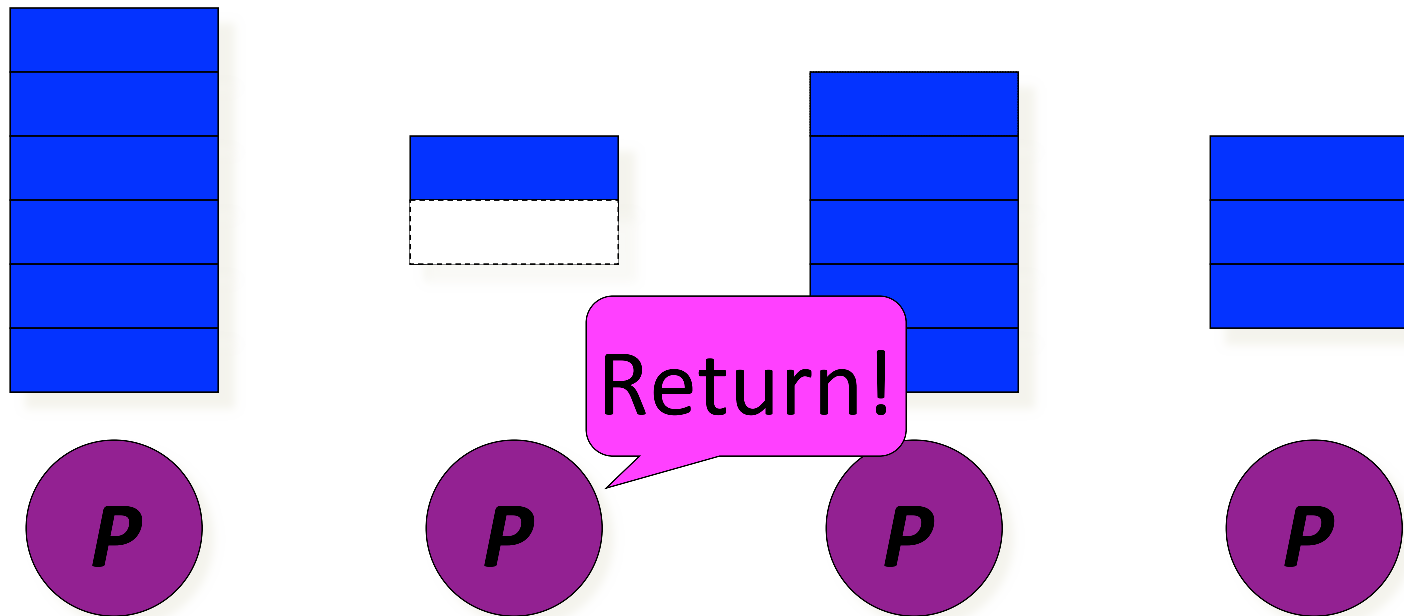
Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



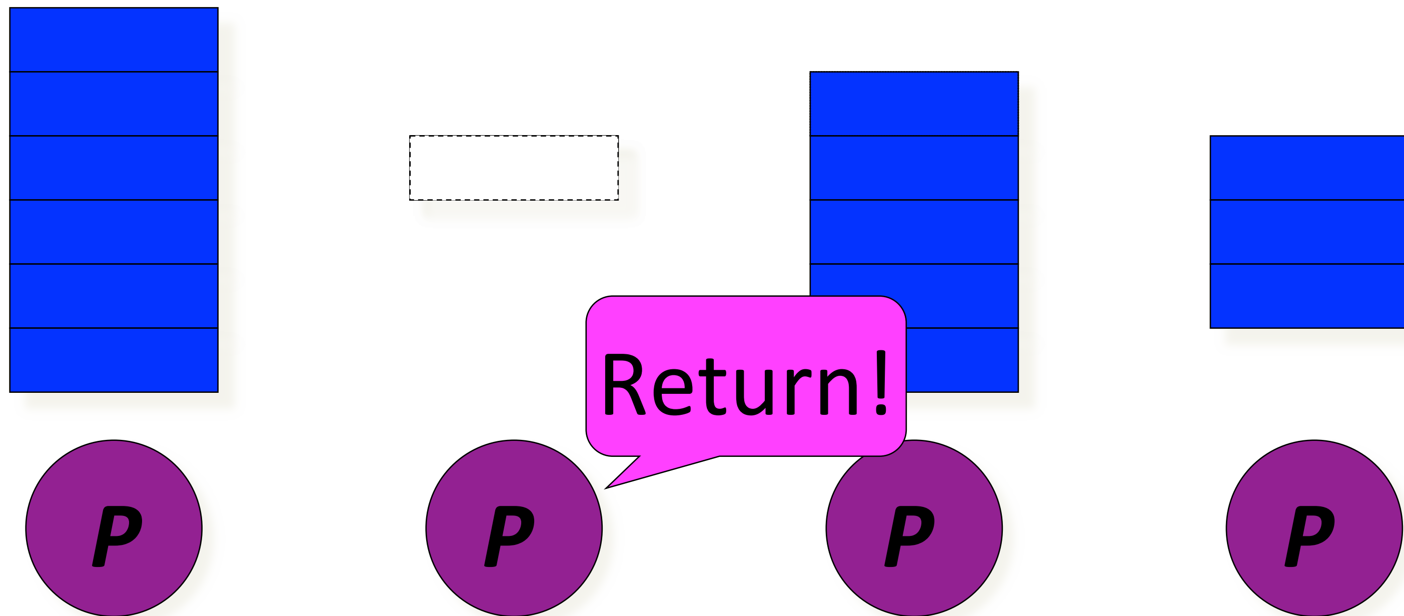
Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



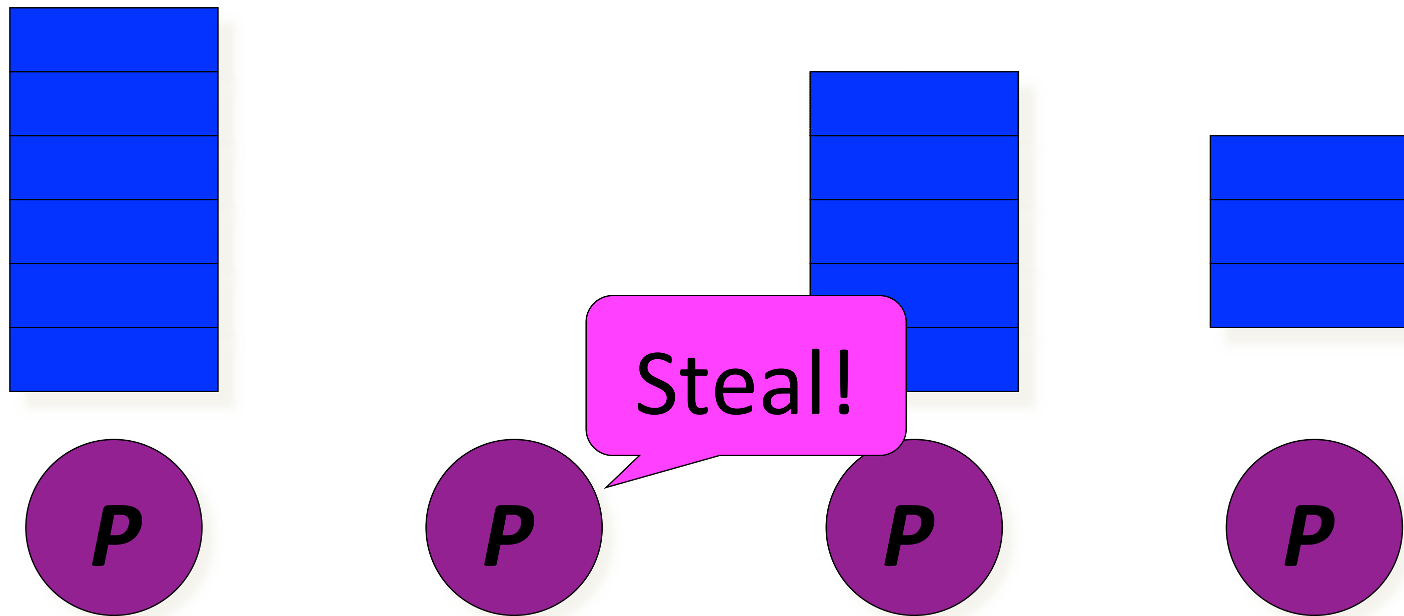
Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.

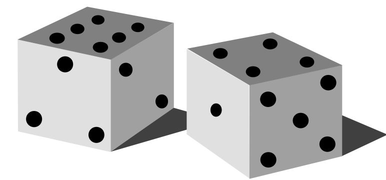


Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.

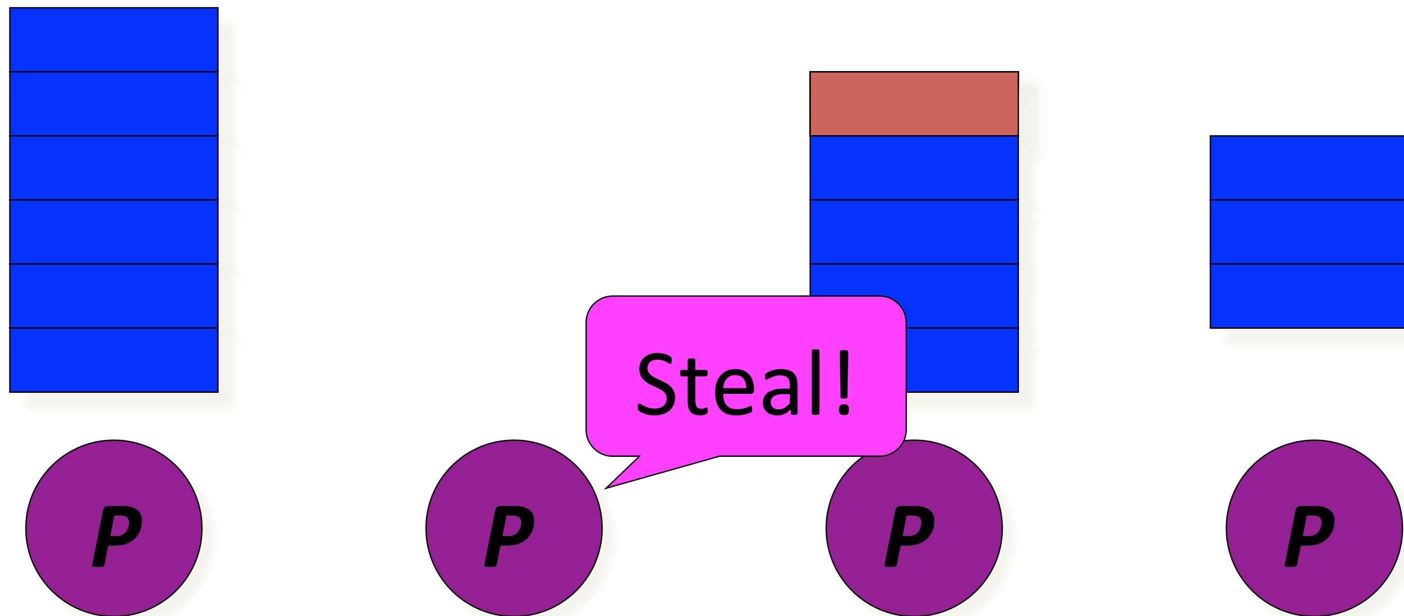


When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.

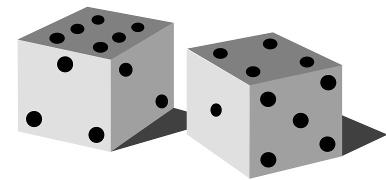


Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.

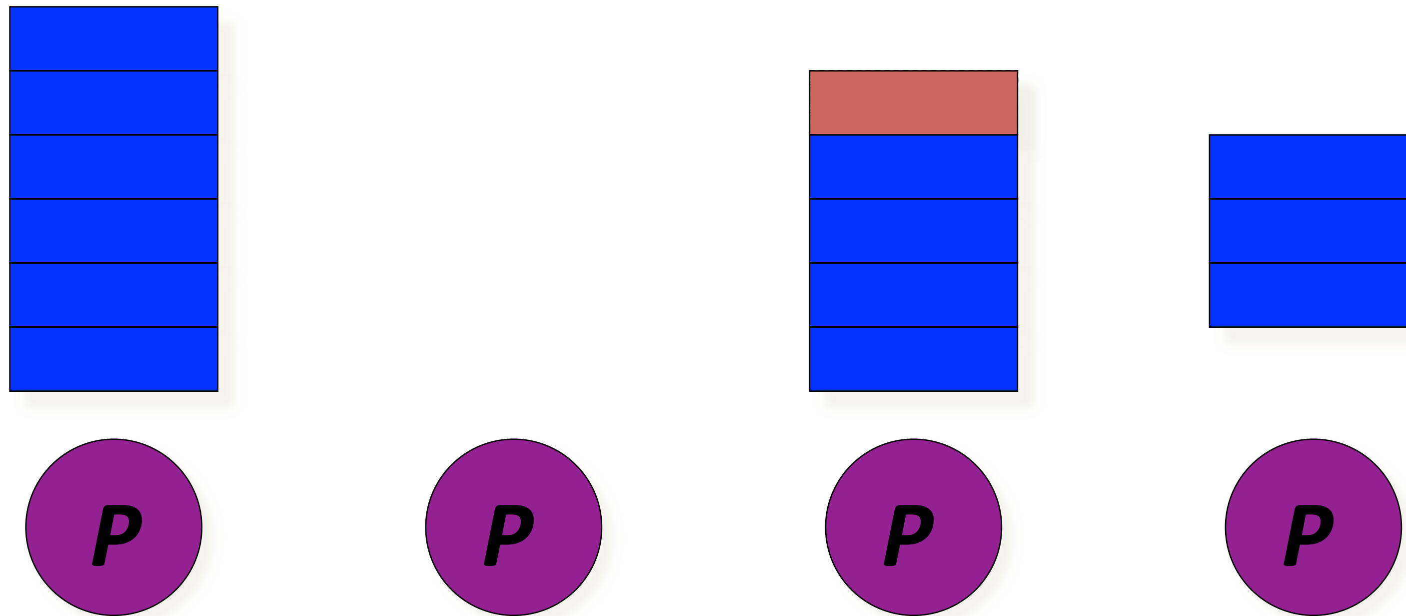


When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.

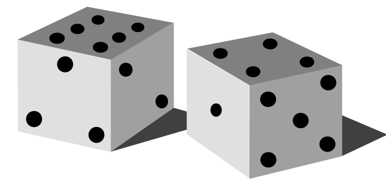


Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.

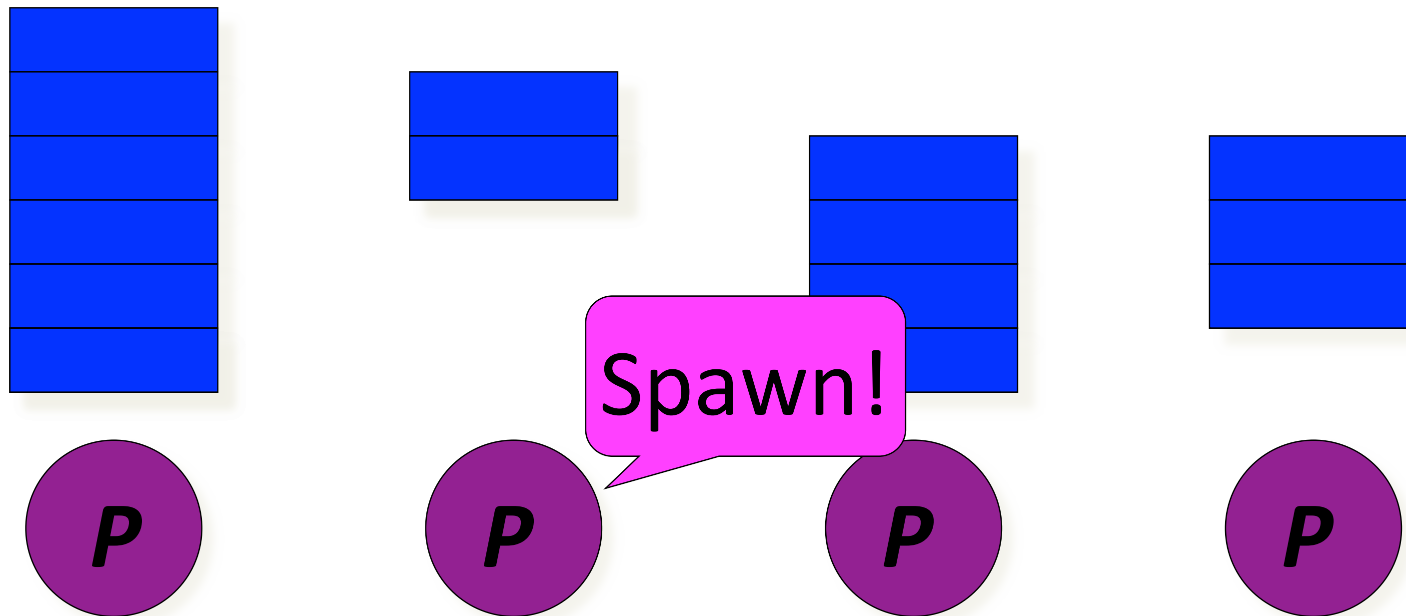


When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.

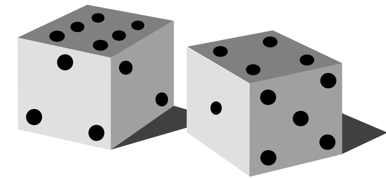


Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.



Intel TBB
thread building blocks

TBB = cilk + OpenMP + c++

Intel TBB parallel_for

```
void operator() (const blocked_range<int>& r) const {
if (length >= minimum) return 0;
if(hops==n){
  if (length < minimum) {
    MutexType::scoped_lock lock(Mutex);
      if (length < minimum)
          minimum = length;
}
} else {
int pathes[anz*(n-hops+1)];
 int j=0;
 for(int i = r.begin(); i != r.end(); i++) {
     if (!present(i, hops, path)) {
       int * path = &pathes[j*anz];
j++;
 memcpy(path,new_path,(hops+1)*sizeof(int));
int me = path[hops-1];
int dist = distanceArray[me*anz+i];
int newLength = length + dist;
path[hops]=i;
int grain=hops==0?1:hops*2;
parallel_for(blocked_range<int>(1,n,grain), TSP
(newLength,hops+1,path));
} } //End operator()
}
}
```

Intel TBB
tâches

```
task * execute() {
int ptr[anz];
if(this->is_stolen_task()) {
  was_stolen = 1;
  memcpy(ptr, new_path, (hops-1) * sizeof(int));
  new_path = ptr;
}
if (length >= minimum) return;
new_path[hops-1]=current;
if(hops == n) {
        if (length < minimum) {
        MutexType::scoped_lock lock(Mutex);
        if(length < minimum)
                minimum = length;
}
} else {
  int count = n-hops;
  set_ref_count(1+count);
   int j=0;
  for(int i=1;i<n;i++){
     if (!present(i, hops, new_path)) {
           j++;
           int dist = distanceArray[current+i*30];
           int tmp = length + dist;
           TSPTask&a =*new(task::allocate_child())
                   TSPTask(tmp,hops+1,new_path,i);
        j < count ? spawn(a) :
                           spawn_and_wait_for_all(a);
  }
}
```

TBB
parallel_for vs task (8 cœurs)

Version	15 Städte	16 Städte	17 Städte
sequentiell	5,75 s	30,68 s	130,46 s
parallel_for	2,31 s	11,38 s	46,39 s
Speedup	2,49	2,70	2,81
Effektivität	0,31	0,34	0,35

Version	15 Städte	16 Städte	17 Städte
sequentiell	5,75 s	30,68 s	130,46 s
task	1,76 s	9,98 s	45,10 s
Speedup	3,26	3,07	2,89
Effizienz	0,41	0,38	0,36

Version	15 Städte	16 Städte	17 Städte
sequentiell	5,75 s	30,68 s	130,46 s
task + stolen	1,66 s	9,65 s	41,39 s
Speedup	3,46	3,18	3,15
Effizienz	0,43	0,40	0,39

```
      minimum = length;
    }
  }
  else {
    int count = n-hops;
    set_ref_count(1+count);
    int j = 0;

    for (int i=1; i < n; i++) {
      if (!present(i, hops, new_path)) {
        j++;

        int dist = distanceArray[current+i*30];
        int tmp = length + dist;

        TSPTask& a = *new(task::allocate_child() ) TSPTask(tmp,hops+1,new_path,i);
        j < count ? spawn(a) : spawn_and_wait_for_all(a);
      }
    }
  }
} //End operator()
```

TBB -TSP
nombre de tâches volées (16 villes)

Niveau	Tasks	davon gestohlen	in %
1	15	14	93
2	210	50	24
3	2730	41	1,5
4	32760	63	0,2
5	356213	117	~0
6	3109590	146	~0
7	18182241	91	~0
8	64528584	62	~0
9	133033964	24	~0
10	153320304	7	~0
11	96375125	6	~0
12	32810436	0	-
13	5968656	0	-
14	552590	0	-
15	17144	0	-

```
 return;

/* same as always here */
[...]
}
```

TBB –TSP
Cut-off

Passer sur code séquentiel à partir d'un seuil

Cut-Off	Laufzeiten in Sekunden		
	15 Städte	16 Städte	17 Städte
1	6,25	32,20	133,09
2	1,06	5,06	19,59
3	0,71	4,14	15,70
4	0,64	3,26	14,86
5	0,62	3,49	14,18
6	0,58	3,55	15,29
7	0,65	3,66	16,21
8	0,73	3,83	15,00
9	0,94	4,64	18,75
10	1,34	6,47	22,42
11	2,09	8,39	32,05
12	2,14	10,29	38,96
13	1,99	10,68	40,78
14	2,09	11,14	45,90
15		10,98	44,10
16			48,31

Version
Sequentiell

```
return;

/* same as always here */
[...]
```

TBB –TSP
Cut-off

Passer sur code séquentiel à partir d'un seuil

Version	15 Städte	16 Städte	17 Städte
Sequentiell	5,75 s	30,68 s	130,46 s
task + cut	0,58 s	3,26 s	14,18 s
Speedup	9,91	9,41	9,20

Speed-up super linéaire

Cut-Off	Laufzeiten in Sekunden		
	15 Städte	16 Städte	17 Städte
1	6,25	32,20	133,09
2	1,06	5,06	19,59
3	0,71	4,14	15,70
4	0,64	3,26	14,86
5	0,62	3,49	14,18
6	0,58	3,55	15,29
7	0,65	3,66	16,21
8 20	0,73	3,83	15,00
9	0,94	4,64	18,75
10	1,34	6,47	22,42
11	2,09	8,39	32,05
12	2,14	10,29	38,96
13	1,99	10,68	40,78
14	2,09	11,14	45,90
15		10,98	44,10
16			48,31

Version
Sequentiell

Comparaison OpenMP, Cilk, TBB

- OpenMP
  - Facilité d'emploi
  - Annotation du code
  - Tâches = bricolage
- Cilk
  - Élégance de l'approche – efficacité du vol de travail
  - Optimisation de l'exécution s'il n'y a pas de vol
  - Modèle de performance
- TBB
  - Puissance du C++
  - Lourd à mettre en œuvre

The Free Lunch Is Over

- Les performances ne viennent pas sans
  - Adapter / repenser les algorithmes
  - Réduire la contention
  - Tenir compte de l'architecture