Programação Paralela com OpenMP

ELC139 - Programação Paralela

João Vicente Ferreira Lima (UFSM)

Universidade Federal de Santa Maria
jvlima@inf.ufsm.br
http://www.inf.ufsm.br/~jvlima

2023/1

Outline

Outline

Task parallelism

Task parallelism

- **Task parallelism** or functional parallelism or control parallelism.
- Decomposes the computation rather than the manipulated data.
  - Programming model for tasks that perform different computations.
- Ex.: Cilk, Intel TBB, OpenMP.

Task dependency

- Tasks with dependencies can unfold a directed acyclic graph (DAG).
  - Expressed by synchronization such as `sync` keyword.
- If data dependencies are considered, the algorithm unfolds a data flow graph (DFG).
- Ex.: Jade, Athapascan, OpenMP (new), KAAPI/XKaapi, StarPU, OmpSs, Intel Offload.

Task parallelism

Task parallelism

- **Task parallelism** or functional parallelism or control parallelism.
- Decomposes the computation rather than the manipulated data.
  - Programming model for tasks that perform different computations.
- Ex.: Cilk, Intel TBB, OpenMP.

Task dependency

- Tasks with dependencies can unfold a directed acyclic graph (DAG).
  - Expressed by synchronization such as `sync` keyword.
- If data dependencies are considered, the algorithm unfolds a data flow graph (DFG).
- Ex.: Jade, Athapascan, OpenMP (new), KAAPI/XKaapi, StarPU, OmpSs, Intel Offload.

OpenMP tasks

Task construct

```
#pragma omp task
```

Barrier taskwait

```
#pragma omp taskwait
```

- Independent units of work.
- Recursive tasks.
- Unfold parallelism at runtime.
- The OpenMP implementation decides when/where to execute:
  - Immediately (in depth, depth-first or work-first)
  - Latter (in breadth, breadth-first or help-first)

OpenMP tasks

Task construct

```
#pragma omp task
```

Barrier `taskwait`

```
#pragma omp taskwait
```

- Independent units of work.
- Recursive tasks.
- Unfold parallelism at runtime.
- The OpenMP implementation decides when/where to execute:
  - Immediately (in depth, depth-first or work-first)
  - Latter (in breadth, breadth-first or help-first)

OpenMP tasks

Task construct

```
#pragma omp task
```

Barrier `taskwait`

```
#pragma omp taskwait
```

- Independent units of work.
- Recursive tasks.
- Unfold parallelism at runtime.
- The OpenMP implementation decides when/where to execute:
  - Immediately (in depth, depth-first or work-first)
  - Latter (in breadth, breadth-first or help-first)

OpenMP tasks

Linked list

```
node* p = head;
while(p) {
  process(p);
  p = p->next;
}
```

informática
Universidade Federal de Santa Maria

OpenMP tasks

Linked list - parallel version

```
#pragma omp parallel
{
#pragma omp single
  {
    node* p = head;
    while(p) {
#pragma omp task firstprivate(p)
      process(p);
      p = p->next;
    }
  }
}
```

informática
Universidade Federal de Santa Maria

Universidade Federal de Santa Maria
SEDES SAPIENTIÆ
1960

OpenMP tasks

Cálculo de Fibonacci

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

Paralelismo de Tarefas

Cálculo de Fibonacci com OpenMP

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

Correto?

Não pois x e y são privados fora do escopo das tarefas.

informática
Universidade Federal de Santa Maria

Cálculo de Fibonacci com OpenMP

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

Correto?

Não pois x e y são privados fora do escopo das tarefas.

Paralelismo de Tarefas

Cálculo de Fibonacci com OpenMP

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

Correto?

Não pois x e y são privados fora do escopo das tarefas.

informática
Universidade Federal de Santa Maria

Paralelismo de Tarefas

Cálculo de Fibonacci com OpenMP

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

Agora sim

Necessitamos dos dois valores no cálculo.

informática
Universidade Federal de Santa Maria

Outline

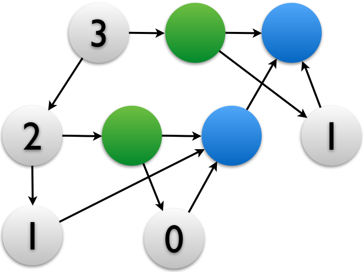
Data flow dependency

Concept

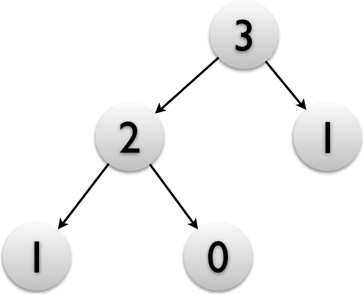
Similar to a DAG of tasks, but combines **task dependencies with data-driven execution**.

- Execution is controlled by the **Data Flow Graph (DFG)**.
- Unlike the program recursion structure of **full strict model**.

Fully strict mode (Cilk)



Data flow graph



Data flow dependency

Data access modes

- Read only (**RO or R**) - only read, no permission to modify.
- Write only (**WO or W**) - only write, no wait for data inputs.
- Read and Write (**RW**) - or exclusive mode, read and write.
- Cumulative Write (**CW**) - concurrent write and cumulative.

informática
Universidade Federal de Santa Maria

Universidade Federal de Santa Maria
SEDES SAPIENTIÆ
1960

Data flow dependency

Data access modes

- Read only (**RO or R**) - only read, no permission to modify.
- Write only (**WO or W**) - only write, no wait for data inputs.
- Read and Write (**RW**) - or exclusive mode, read and write.
- Cumulative Write (**CW**) - concurrent write and cumulative.

Data flow dependency

Data access modes

- Read only (**RO or R**) - only read, no permission to modify.
- Write only (**WO or W**) - only write, no wait for data inputs.
- Read and Write (**RW**) - or exclusive mode, read and write.
- Cumulative Write (**CW**) - concurrent write and cumulative.

Data flow dependency

Data access modes

- Read only (**RO or R**) - only read, no permission to modify.
- Write only (**WO or W**) - only write, no wait for data inputs.
- Read and Write (**RW**) - or exclusive mode, read and write.
- Cumulative Write (**CW**) - concurrent write and cumulative.

Data flow dependency

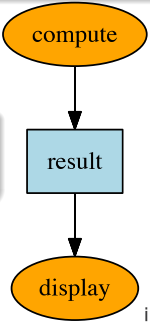
The concepts here are the same from computer architecture in which we can replace **instruction** by **task**.

Data dependencies

- Task t_i produces a result that may be used by a task t_j.
- Task t_j is data dependent on task t_k, and t_k is data dependent on t_i ($t_i \rightarrow t_k \rightarrow t_j$).

```
compute( input, &result );

display( &result );
```



Data flow dependency

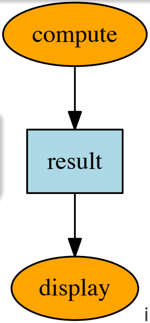
The concepts here are the same from computer architecture in which we can replace **instruction** by **task**.

Data dependencies

- Task t_i produces a result that may be used by a task t_j.
- Task t_j is data dependent on task t_k, and t_k is data dependent on t_i $(t_i \rightarrow t_k \rightarrow t_j)$.

```
compute( input, &result );

display( &result );
```



informática
Universidade Federal de Santa Maria

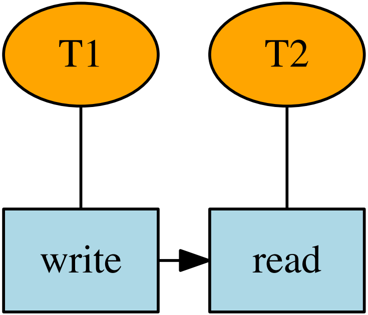
Data hazards

Data hazards

There is a **dependence** between task and the we must preserve the execution order.

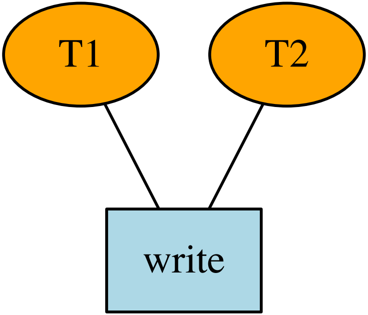
- **RAW** (*Read after Write*).
- **WAW** (*Write after Write*).
- **WAR** (*Write after Read*).

Data flow dependency



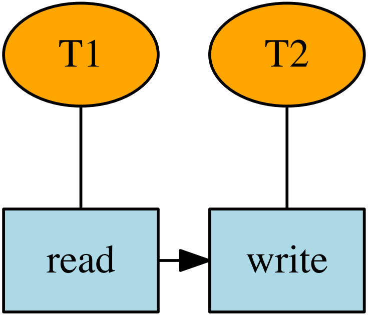
- **RAW** (*Read after Write*) - or true dependency, data dependency.
- A task depends on the result produced by a previous task.

Data flow dependency



- **WAW** (*Write after Write*) - or output dependency.
- The execution order will affect the final output.

Data flow dependency



- **WAR** (*Write after Read*) - or anti-dependency.
- A task writes a value before it is read.

Data flow example

In our next examples, we will use the following keywords:

- **in** - read access.
- **out** - write access.
- **inout** - read and write access.

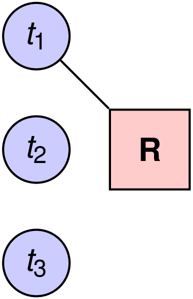
```
void reading(in int a) {}
void modifying(inout int b) {}
main(void)
{
  int a;
  reading( a );
  reading( a );
  reading( a );
  modifying( a );
  modifying( a );
}
```

Data flow example

```
void reading(in int a) {}
void modifying(inout int b) {}
main(void)
{
  int a;
  reading( a );
  reading( a );
  reading( a );
  modifying( a );
  modifying( a );
}
```

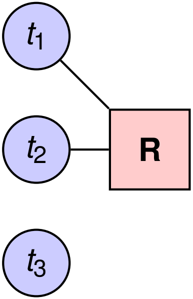
Data flow example

```
void reading(in int a) {}
void modifying(inout int b) {}
main(void)
{
  int a;
  reading( a );
  reading( a );
  reading( a );
  modifying( a );
  modifying( a );
}
```



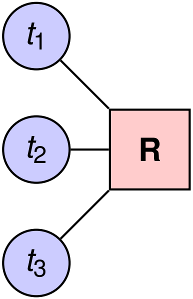
Data flow example

```
void reading(in int a) {}
void modifying(inout int b) {}
main(void)
{
  int a;
  reading( a );
  reading( a );
  reading( a );
  modifying( a );
  modifying( a );
}
```



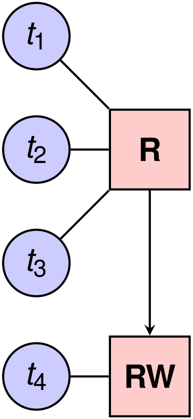
Data flow example

```
void reading(in int a) {}
void modifying(inout int b) {}
main(void)
{
  int a;
  reading( a );
  reading( a );
  reading( a );
  modifying( a );
  modifying( a );
}
```



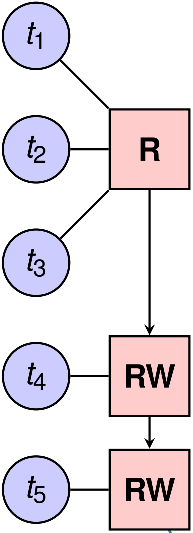
Data flow example

```
void reading(in int a) {}
void modifying(inout int b) {}
main(void)
{
  int a;
  reading( a );
  reading( a );
  reading( a );
  modifying( a );
  modifying( a );
}
```



Data flow example

```
void reading(in int a) {}
void modifying(inout int b) {}
main(void)
{
  int a;
  reading( a );
  reading( a );
  reading( a );
  modifying( a );
  modifying( a );
}
```



Fibonacci example

```
void fibo( int n, int* res )
{
  int x, y;
  if( n < 2 ){
    *res = n;
  } else {
    fibo( n-1, &x );
    fibo( n-2, &y );
    *res = x+y;
  }
}

int main(void) {
  int n = 3, res;
  fibo( n, &res );
  print( res );
  return 0;
}
```

informática
Universidade Federal de Santa Maria

Fibonacci example

Notice

This recursive Fibonacci is not the best implementation, but it serves our purposes.

Dependency example (again)

In our next example, we will use the following keywords:

- **in** - read access.
- **out** - write access.
- **inout** - read and write access.
- **cout** - cumulative write with global reduction.

Fibonacci example

Notice

This recursive Fibonacci is not the best implementation, but it serves our purposes.

Dependency example (again)

In our next example, we will use the following keywords:

- **in** - read access.
- **out** - write access.
- **inout** - read and write access.
- **cout** - cumulative write with global reduction.

Fibonacci example

```
void fibo( in int n, out int* res )
{
  int x, y;
  if( n < 2 ){
    *res = n;
  } else {
    fibo( n-1, &x );
    fibo( n-2, &y );
    *res = x+y;
  }
}

int main(void) {
  int n = 3, res;
  fibo( n, &res );
  print( res );
  return 0;
}
```

Fibonacci example

```
void fibo( in int n, out int* res )
{
  int x, y;
  if( n < 2 ){
    *res = n;
  } else {
    fibo( n-1, &x );
    fibo( n-2, &y );
    *res = x+y;
  }
}

int main(void) {
  int n = 3, res;
  fibo( n, &res );
  print( res );
  return 0;
}
```

informática
Universidade Federal de Santa Maria

Universidade Santa Maria
SEDES SAPIENTIÆ
1960

Fibonacci example

Previous Fibonacci example

```
} else {
  fibo( n-1, &x );
  fibo( n-2, &y );
  *res = x+y;
}
```

Synchronization problem

If our tasks execute in parallel, we would like **to wait** for the results from the previous two `fibo` tasks.

Solution

1. An explicit synchronization (Cilk's style).
2. A task that depends on the results from the two `fibo` tasks.

Fibonacci example

Previous Fibonacci example

```
} else {
  fibo( n-1, &x );
  fibo( n-2, &y );
  *res = x+y;
}
```

Synchronization problem

If our tasks execute in parallel, we would like **to wait** for the results from the previous two `fibo` tasks.

Solution

1. An explicit synchronization (Cilk's style).
2. A task that depends on the results from the two `fibo` tasks.

Fibonacci example

Previous Fibonacci example

```
} else {
  fibo( n-1, &x );
  fibo( n-2, &y );
  *res = x+y;
}
```

Synchronization problem

If our tasks execute in parallel, we would like **to wait** for the results from the previous two `fibo` tasks.

Solution

1. An explicit synchronization (Cilk's style).
2. A task that depends on the results from the two `fibo` tasks.

Fibonacci example

```
void sum( out int* res, in int x, in int y )
{
  *a = x + y;
}

void fibo( in int n, out int* res )
{
  int x, y;
  if( n < 2 ){
    *res = n;
  } else {
    fibo( n-1, &x );
    fibo( n-2, &y );
    sum( res, x, y );
  }
}
```

informática
Universidade Federal de Santa Maria

Universidade Federal de Santa Maria
SEDES SAPIENTIÆ
1960

Fibonacci example

```
void sum( out int* res, in int x, in int y )
{
  *a = x + y;
}

void fibo( in int n, out int* res )
{
  int x, y;
  if( n < 2 ){
    *res = n;
  } else {
    fibo( n-1, &x );
    fibo( n-2, &y );
    sum( res, x, y );
  }
}
```

informática
Universidade Federal de Santa Maria

Universidade Federal de Santa Maria
SEDES SAPIENTIÆ
1960

Fibonacci example (cumulative)

```
void fibo( in int n, cout int* res )
{
  int x, y;
  if( n < 2 ){
    *res += n;
  } else {
    fibo( n-1, &x );
    fibo( n-2, &y );
  }
}
```

Fibonacci example (cumulative)

```
void fibo( in int n, cout int* res )
{
  int x, y;
  if( n < 2 ){
    *res += n;
  } else {
    fibo( n-1, &x );
    fibo( n-2, &y );
  }
}
```

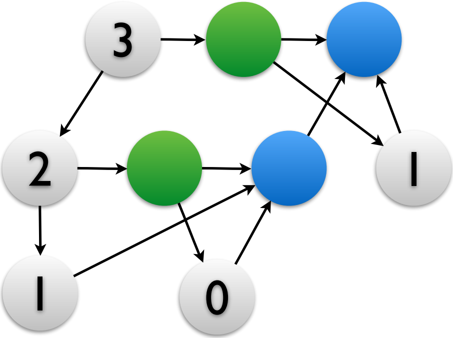
informática
Universidade Federal de Santa Maria

Universidade Federal de Santa Maria
SEDES SAPIENTIÆ
1960

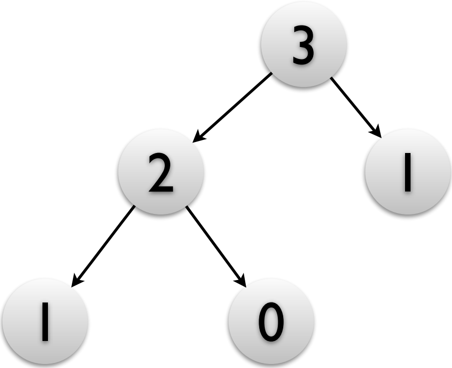
Data flow graph

- Data flow graph (DFG) combines task dependencies with data driven execution.

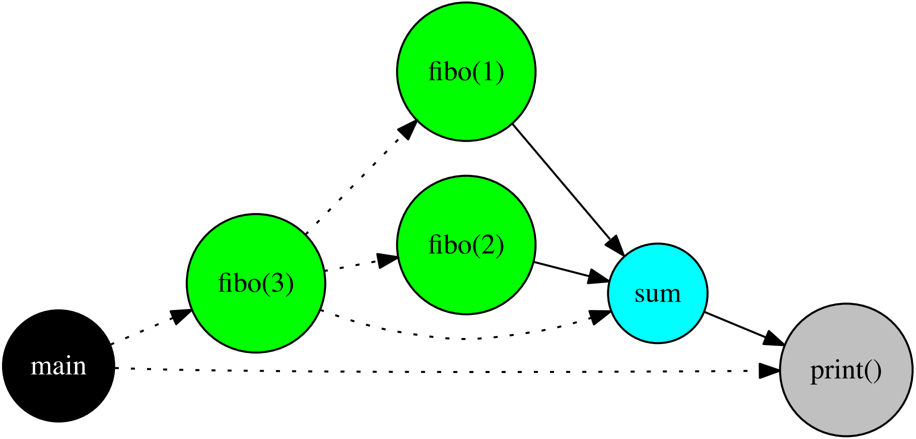
Fully strict mode (Cilk)



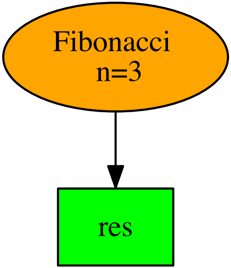
Data flow graph



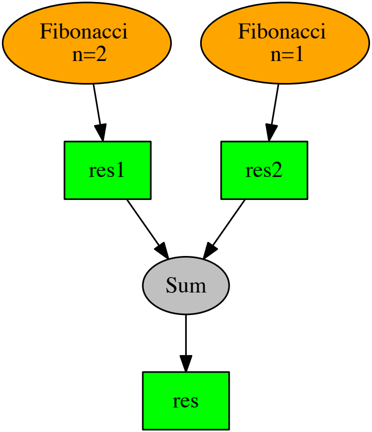
DAG of Fibonacci $n = 3$



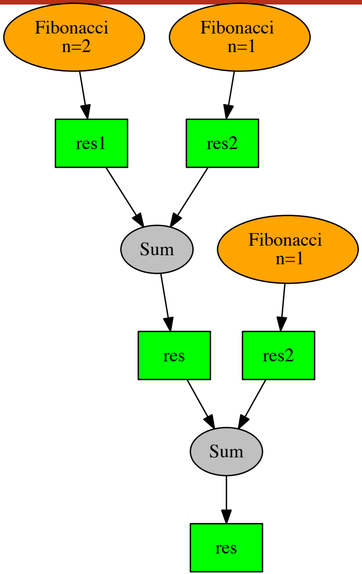
DAG of Fibonacci $n = 3$



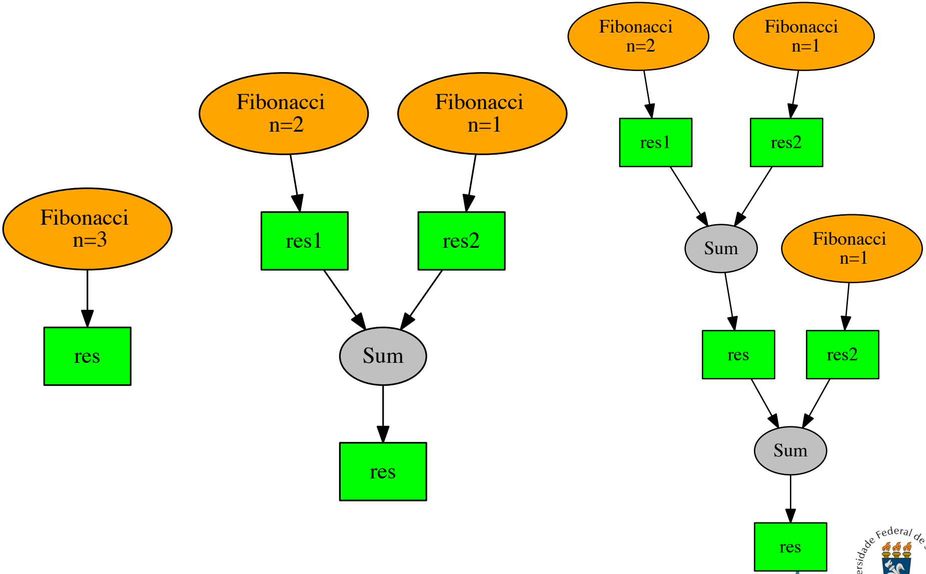
DFG of Fibonacci $n = 3$



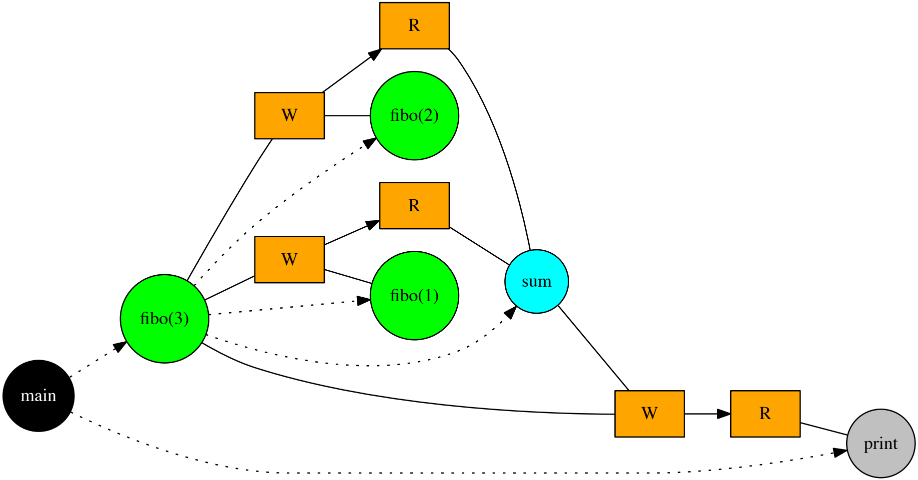
DFG of Fibonacci $n = 3$



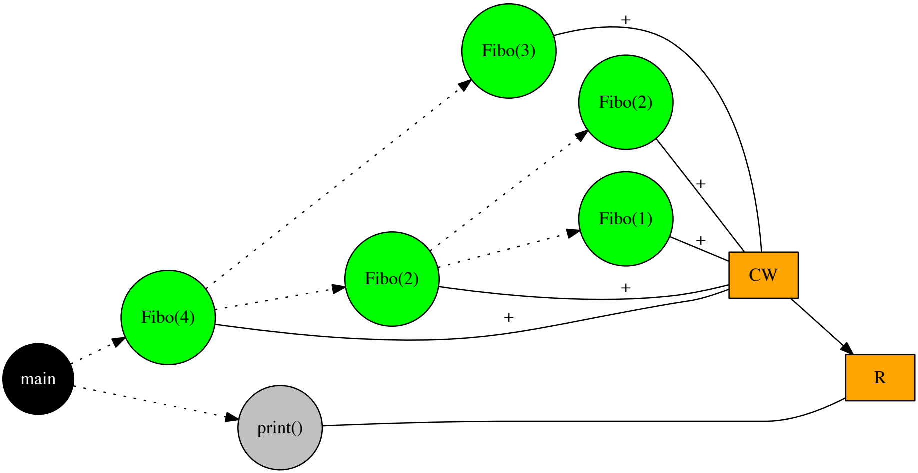
DFG of Fibonacci $n = 3$



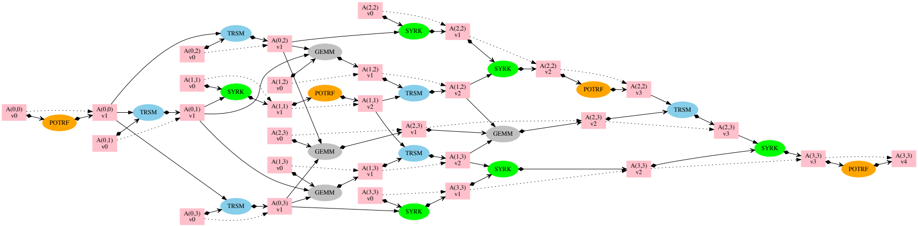
DFG of Fibonacci $n = 3$

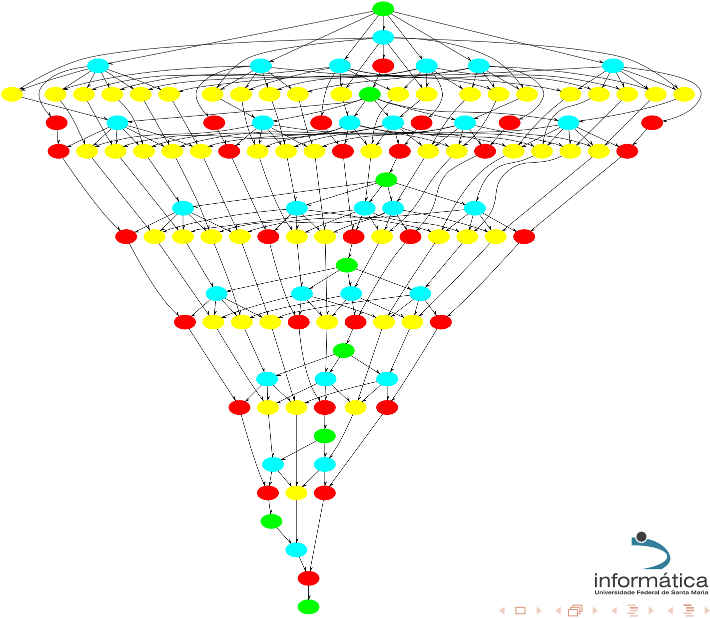


DFG of Fibonacci $n = 3$ (cumulative)

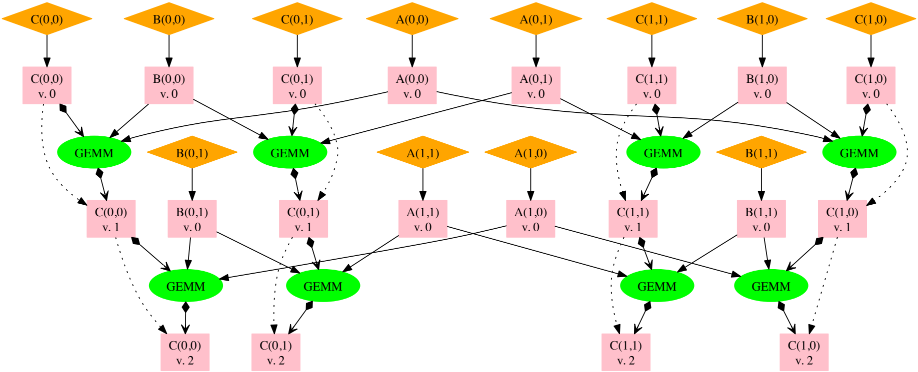


DFG of Cholesky factorization





DFG of Blocked matrix multiplication



informática
Universidade Federal de Santa Maria

Universidade Federal de Santa Maria
SEDES SAPIENTIÆ
1960

https://joao-ufsm.github.io/par2023a/