Scheduling for Large Scale Systems
Knoxville, TN, May 13-15, 2009

Generic Dynamic Scheduler for Numerical Libraries on Multicore Processors

Prepared by Jakub Kurzak & Piotr Łuszczek
Presented by Piotr Łuszczek

Topics

- Scheduling Options
  - PLASMA static scheduling
  - Nested parallelism – Cilk, TBB, OpenMP
  - Declarative programming & tuple spaces
  - Dataflow – SMPSs, GUST
- GUST
  - Motivation
  - GUST API
  - Implementation principles
  - Some internals
  - Discussion of current and planned features

Starting Point

Tile Cholesky Factorization

```
FOR k = 0..TILES-1
    FOR n = 0..k-1
        A[k][k] ← DSYRK(A[k][n], A[k][k])
    A[k][k] ← DPOTRF(A[k][k])
    FOR m = k+1..TILES-1
        FOR n = 0..k-1
            A[m][k] ← DGEMM(A[k][n], A[m][n], A[m][k])
        A[m][k] ← DTRSM(A[k][k], A[m][k])
```

Tile QR Factorization

```
FOR k = 0..TILES-1
    A[k][k], T[k][k] ← DGRQRT(A[k][k])
    FOR m = k+1..TILES-1
        A[k][k], A[m][k], T[m][k] ← DTSQRT(A[k][k], A[m][k], T[m][k])
    FOR n = k+1..TILES-1
        A[k][n] ← DLARFB(A[k][k], T[k][k], A[k][n])
        FOR m = k+1..TILES-1
            A[k][n], A[m][n] ← DSSRFB(A[m][k], T[m][k], A[k][n], A[m][n])
```

"Starting Point" is sequential (at best recursive).

PLASMA's Static Scheduling

```
FOR k = 0..TILES-1
   FOR n = 0..k-1
      A[k][k] ← DSYRK(A[k][n], A[k][k])
   A[k][k] ← DPOTRF(A[k][k])
   FOR m = k+1..TILES-1
      FOR n = 0..k-1
         A[m][k] ← DGEMM(A[k][n], A[m][n], A[m][k])
      A[m][k] ← DTRSM(A[k][k], A[m][k])
```

```
k = 0; m = my_core_id;
while (m >= TILES) {
   k++; m = m-TILES+k;
} n = 0;

while (k < TILES && m < TILES) {
   next_n = n; next_m = m; next_k = k;

   next_n++;
   if (next_n > next_k) {
      next_m += cores_num;
      while (next_m >= TILES && next_k < TILES) {
         next_k++; next_m = next_m-TILES+next_k;
      } next_n = 0;
   }

   if (m == k) {
      if (n == k) {
         dpotrf(A[k][k]);
         core_progress[k][k] = 1;
      }
      else {
         while(core_progress[k][n] != 1);
         dsyrk(A[k][n], A[k][k]);
      }
   }
   else {
      if (n == k) {
         while(core_progress[k][k] != 1);
         dtrsm(A[k][k], A[m][k]);
         core_progress[m][k] = 1;
      }
      else {
         while(core_progress[k][n] != 1);
         while(core_progress[m][n] != 1);
         dgemm(A[k][n], A[m][n], A[m][k]);
      }
   }
   n = next_n; m = next_m; k = next_k;
}
```

Cholesky Factorization

- Not too complex for Cholesky, LU, QR
- Accommodates for data locality / reuse
- The fastest we know of on shared memory
- Possibly applicable to distributed memory

- Only applicable to simple cases
- Hard to develop
- Slightly different for each case

~~Nested~~ Doomed Parallelism

```
FOR k = 0..TILES-1
   A[k][k], T[k][k] ← DGRQRT(A[k][k])
   FOR m = k+1..TILES-1
      A[k][k], A[m][k], T[m][k] ← DTSQRT(A[k][k], A[m][k], T[m][k])
   FOR n = k+1..TILES-1
      A[k][n] ← DLARFB(A[k][k], T[k][k], A[k][n])
      FOR m = k+1..TILES-1
         A[k][n], A[m][n] ← DSSRFB(A[m][k], T[m][k], A[k][n], A[m][n])
```

Tile QR Factorization

- Basically only suitable for recursion
- Not easy at all for other classes of algorithms
- Oblivious to data locality / reuse
- Results in poor schedules → poor performance
- Hard to imagine on distributed memory

```
cilk void qr_panel(int k)
{
   int m;

   dgeqrt(A[k][k], T[k][k]);

   for (m = k+1; m < TILES; m++)
      dtsqrt(A[k][k], A[m][k], T[m][k]);
}

cilk void qr_update(int n, int k)
{
   int m;

   dlarfb(A[k][k], T[k][k], A[k][n]);

   for (m = k+1; m < TILES; m++)
      dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);

   if (n == k+1)
      spawn qr_panel(k+1);
}


spawn qr_panel(0);
sync;

for (k = 0; k < TILES; k++) {
   for (n = k+1; n < TILES; n++)
      spawn qr_update(n, k);
   sync;
}
```

Declarative Programming

```
syrk(k=0..TILES, n=0..k-2, A, T)
→ syrk(k, n+1, ?, T);

syrk(k=0..TILES, k-1, A, T)
→ potrf(k, T);

potrf(k=0..TILES, T)
→ trsm(k, k+1..TILES, T, ?);

gemm(k=0..TILES, m=k+1..TILES, n=0..k-2, A, B, C)
→ gemm(k, m, n+1, ?, ?, C);

gemm(k=0..TILES, m=k+1..TILES, k-2, A, B, C)
→ trsm(k, m, ?, C);

trsm(k=0..TILES, m=k+1..TILES, T, C)
→ syrk(m, k, A, ?),
→ gemm(m, m+1..TILES, k, C, ?, ?),
→ gemm(k+1..m-1, m, k, ?, B, ?);
```

Cholesky Factorization

- Task described using tuple spaces
- Formulas express dependencies

- Strictly local view
- No serialization
- No DAG construction
- Scalable
- Suitable for distributed memory

- Only applicable to simple cases
- Hard to develop

SMPSs

- Sequential algorithm definition → trivial to develop
- Dataflow scheduling → very good schedules
- Dataflow scheduling → great data locality / reuse
- Marginally worse than PLASMA's static schedules

- Possibly applicable to small-scale distributed
- Inherent limitation for large scale (Petascale)

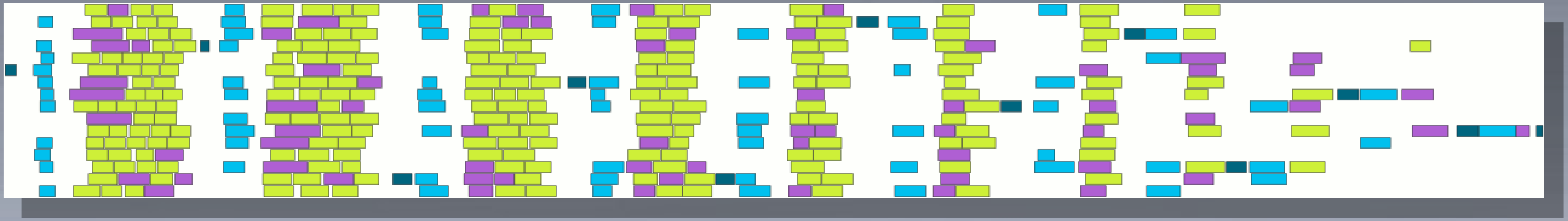
Tile QR Factorization

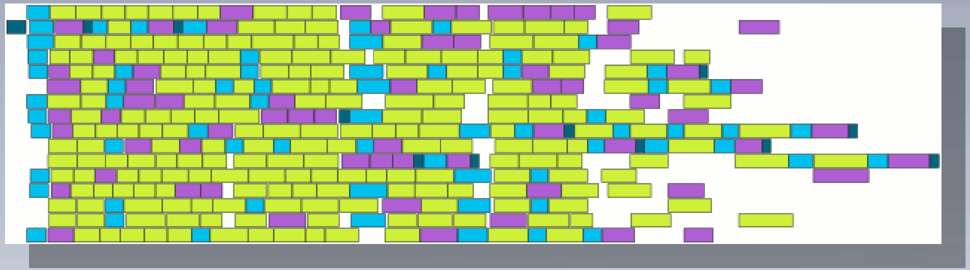
```
FOR k = 0..TILES-1
   A[k][k], T[k][k] ← DGRQRT(A[k][k])
   FOR m = k+1..TILES-1
      A[k][k], A[m][k], T[m][k] ← DTSQRT(A[k][k], A[m][k], T[m][k])
   FOR n = k+1..TILES-1
      A[k][n] ← DLARFB(A[k][k], T[k][k], A[k][n])
      FOR m = k+1..TILES-1
         A[k][n], A[m][n] ← DSSRFB(A[m][k], T[m][k], A[k][n], A[m][n])
```

```
#pragma css task \
   inout(RV1[NB][NB]) output(T[NB][NB])
void dgeqrt(double *RV1, double *T);

#pragma css task \
   inout(R[NB][NB], V2[NB][NB]) ...
void dtsqrt(double *R, double *V2, ...

#pragma css task \
   input(V1[NB][NB], T[NB][NB]) ...
void dlarfb(double *V1, double *T, ...

#pragma css task \
   input(V2[NB][NB], T[NB][NB]) ...
void dssrfb(double *V2, double *T, ...

#pragma css start
for (k = 0; k < TILES; k++) {

   dgeqrt(A[k][k], T[k][k]);

   for (m = k+1; m < TILES; m++)
      dtsqrt(A[k][k], A[m][k], T[m][k]);

   for (n = k+1; n < TILES; n++) {
      dlarfb(A[k][k], T[k][k], A[k][n]);
      for (m = k+1; m < TILES; m++)
         dssrfb(A[m][k], T[m][k], A[k][n], ...
   }
}
#pragma css finish
```

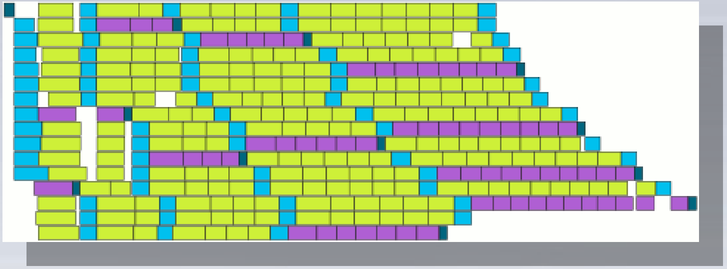
Schedule Comparison

Cilk



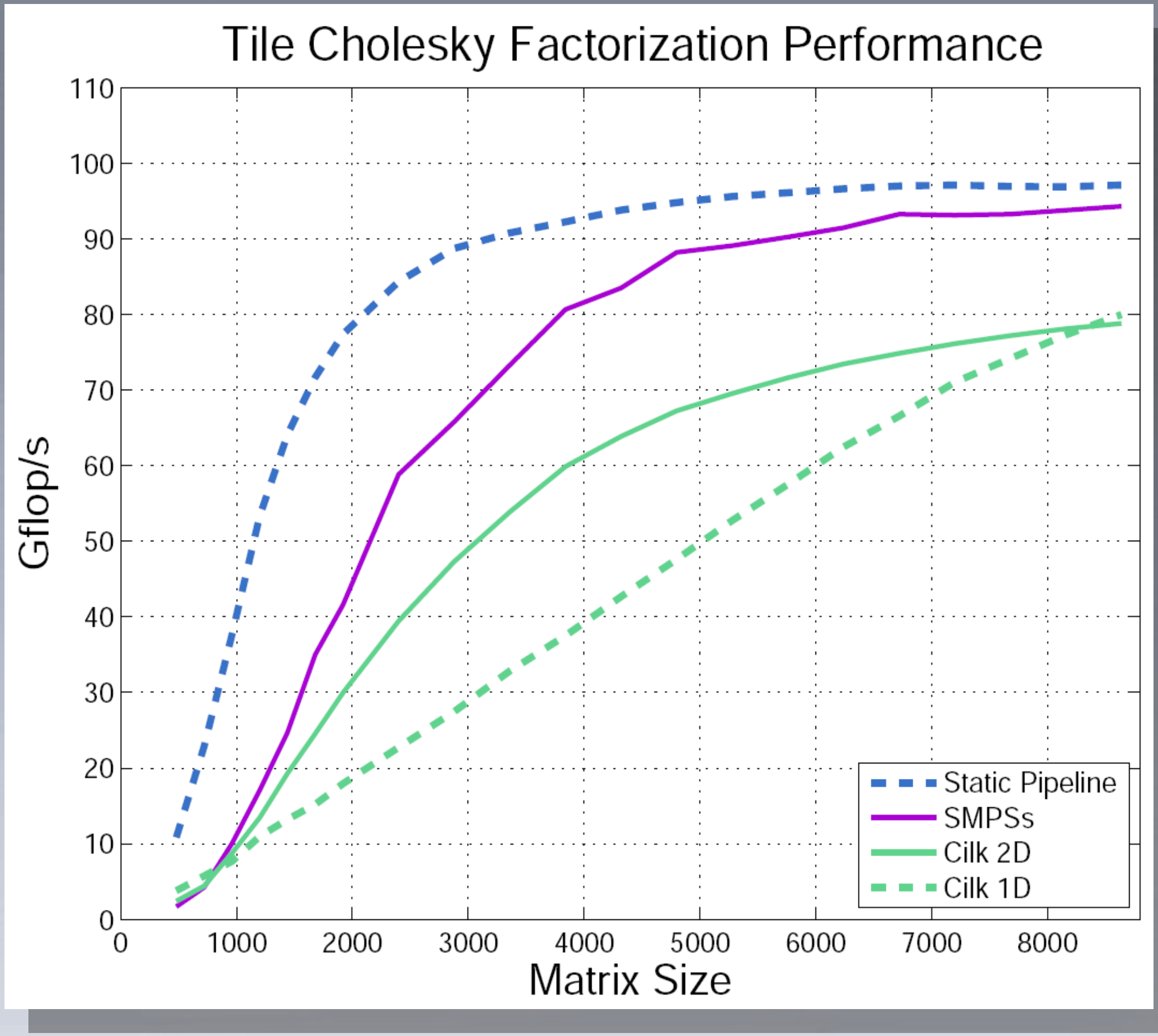


SMPSs



Static Pipeline

Performance Comparison



- static schedule – very good
- SMPSs – somewhat worse
- Cilk – much worse

quad-socket, quad-core Intel Tigertone 2.4 GHz

GUST Motivation

- Follow SMPSs' approach
- Sequential algorithm definition
  - Extreme ease of use → productivity
  - Extremely fast prototyping of new algorithms / ideas
- Dynamic scheduling
  - Compensating for performance fluctuations → better performance
  - Eliminating artificial synchronizations → better throughput
    (e.g., between factorization and solve, steps of iterative refinement, etc.)

- Craft for use in numerical libraries
- Drop compiler support in favor of an API → more robust, more control
- Customize task prioritization
- Customize data renaming

GUST API — defining a tasks

```
void CORE_dgetrf(
    int M, int N, int IB,
    double *A,
    int LDA, int *IPIV)
{
    ...
}
```

In the function implementing the task

- *clear the argument list*
- *declare arguments as local variables*
- *set arguments values using a macro*

```
void CORE_dgetrf()
{
    int M, N IB;
    double *A;
    int LDA, *IPIV;
    unpack_args_6(M, N, IB, A, LDA, IPIV);

    ...
}
```

GUST API — queueing a tasks

```
CORE_dgetrf(
    NB     ,
    NB     ,
    IB     ,
    A(k, k),
    NB     ,
    IPIV(k, k));
```

Create a task instead of calling the function

- *pass arguments by reference**
- *specify sizes*
- *specify directions*
- *finish the list with a NULL*

**Passing of scalar arguments (VALUE) has “pass by value” semantics; A copy is made at the time of inserting the task.*

```
Insert_Task(CORE_dgetrf,
    &NB       , sizeof(int)         , VALUE ,
    &NB       , sizeof(int)         , VALUE ,
    &IB       , sizeof(int)         , VALUE ,
    A(k, k)   , NB*NB*sizeof(double), INOUT ,
    &NB       , sizeof(int)         , VALUE ,
    IPIV(k, k), NB*sizeof(double)   , OUTPUT,
    NULL);
```

GUST API — Tile LU

Tile LU – serial

```
for (k = 0; k < BB; k++) {
    CORE_dgetrf(
        A(k, k),
        IPIV(k, k),

    for (n = k+1; n < BB; n++)
        CORE_dgessm(
            IPIV(k, k),
            A(k, k),
            A(k, n),

    for (m = k+1; m < BB; m++) {
        CORE_dtstrf(
            A(k, k),
            A(m, k),
            L(m, k),
            IPIV(m, k),

        for (n = k+1; n < BB; n++)
            CORE_dssssm(
                A(k, n),
                A(m, n),
                L(m, k),
                A(m, k),
                IPIV(m, k),
    }
}
```

Tile LU – parallel

```
for (k = 0; k < BB; k++) {
    Insert_Task(CORE_dgetrf,
        A(k, k), NB*NB*sizeof(double), INOUT,
        IPIV(k, k), NB*sizeof(double), OUTPUT,

    for (n = k+1; n < BB; n++)
        Insert_Task(CORE_dgessm,
            IPIV(k, k), NB*sizeof(double), INPUT,
            A(k, k), NB*NB*sizeof(double), NODEP,
            A(k, n), NB*NB*sizeof(double), INOUT,

    for (m = k+1; m < BB; m++) {
        Insert_Task(CORE_dtstrf,
            A(k, k), NB*NB*sizeof(double), INOUT,
            A(m, k), NB*NB*sizeof(double), INOUT,
            L(m, k), NB*IB*sizeof(double), OUTPUT,
            IPIV(m, k), NB*sizeof(double), OUTPUT,

        for (m = k+1; m < BB; m++)
            Insert_Task(CORE_dssssm,
                A(k, n), NB*NB*sizeof(double), INOUT,
                A(m, n), NB*NB*sizeof(double), INOUT,
                L(m, k), NB*IB*sizeof(double), INPUT,
                A(m, k), NB*NB*sizeof(double), INPUT,
                IPIV(m, k), NB*sizeof(double), INPUT,
    }
}
```

Scalars removed on both sides for clarity.

GUST API

There is a little bit of copy-pasting involved, but the transition from the sequential code to parallel code takes a few minutes and is basically effortless.

GUST Dependency Resolution

- **RAW** – Read After Write
  - OUTPUT → INPUT
  - “true” dependency
  - wait until data is produced
  - majority of dependencies in PLASMA

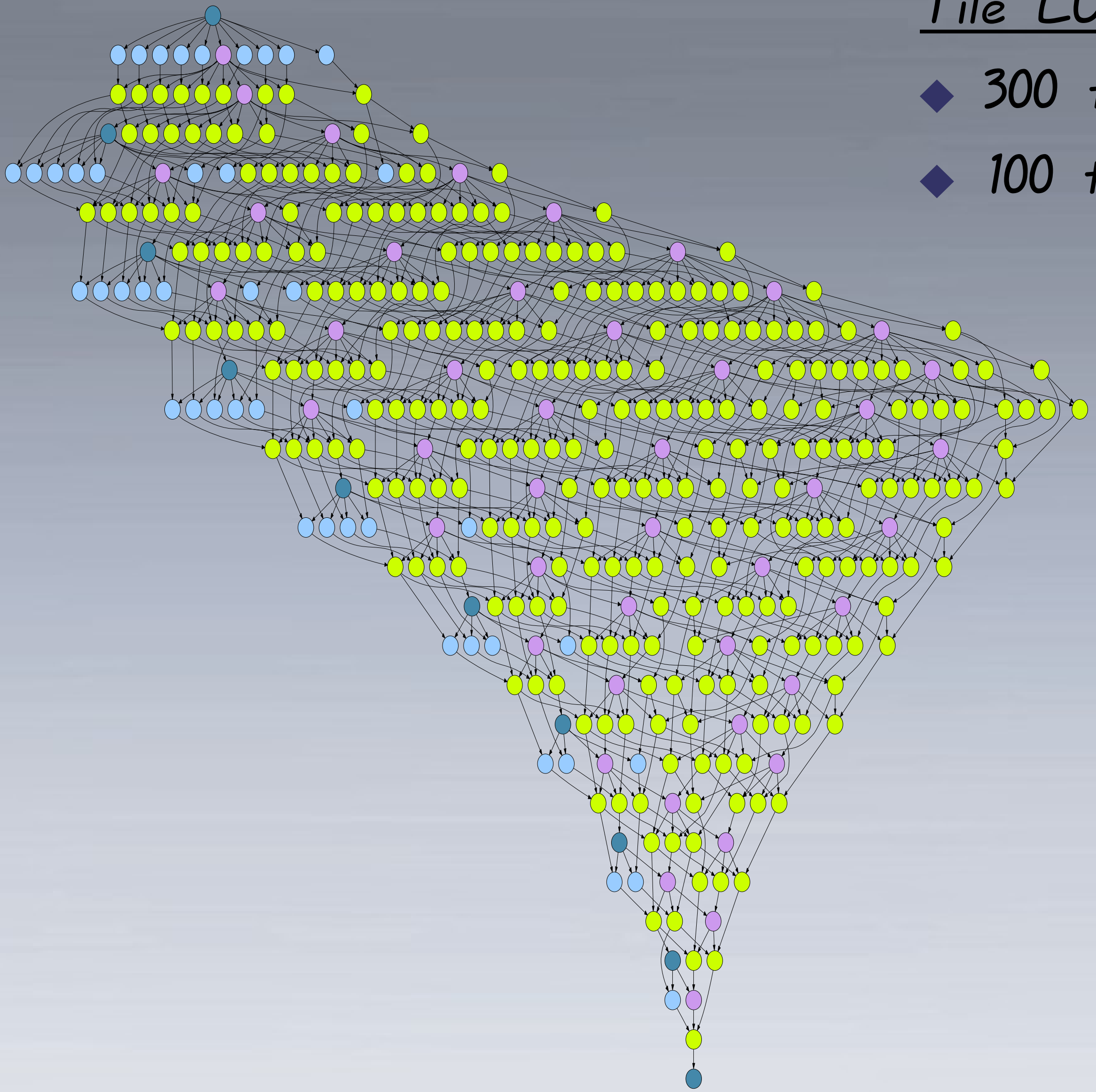
- **WAR** – Write After Read
  - INPUT → OUTPUT
  - can be eliminated with renaming
  - don't overwrite until predecessors done reading
  - likely to occur in PLASMA, but infrequently

- **WAW** – Write After Write
  - OUTPUT → OUTPUT
  - can be eliminated with renaming
  - don't overwrite until predecessors done writing
  - extremely unlikely to happen in PLASMA

GUST Implementation Principles

- Constrained use of resources
  (imagine a hardware implementation)
- Little to none dynamic data structures
- Little to none dynamic memory allocation
- Lightweight synchronization
  - *volatile* where possible
  - *mutex* where necessary

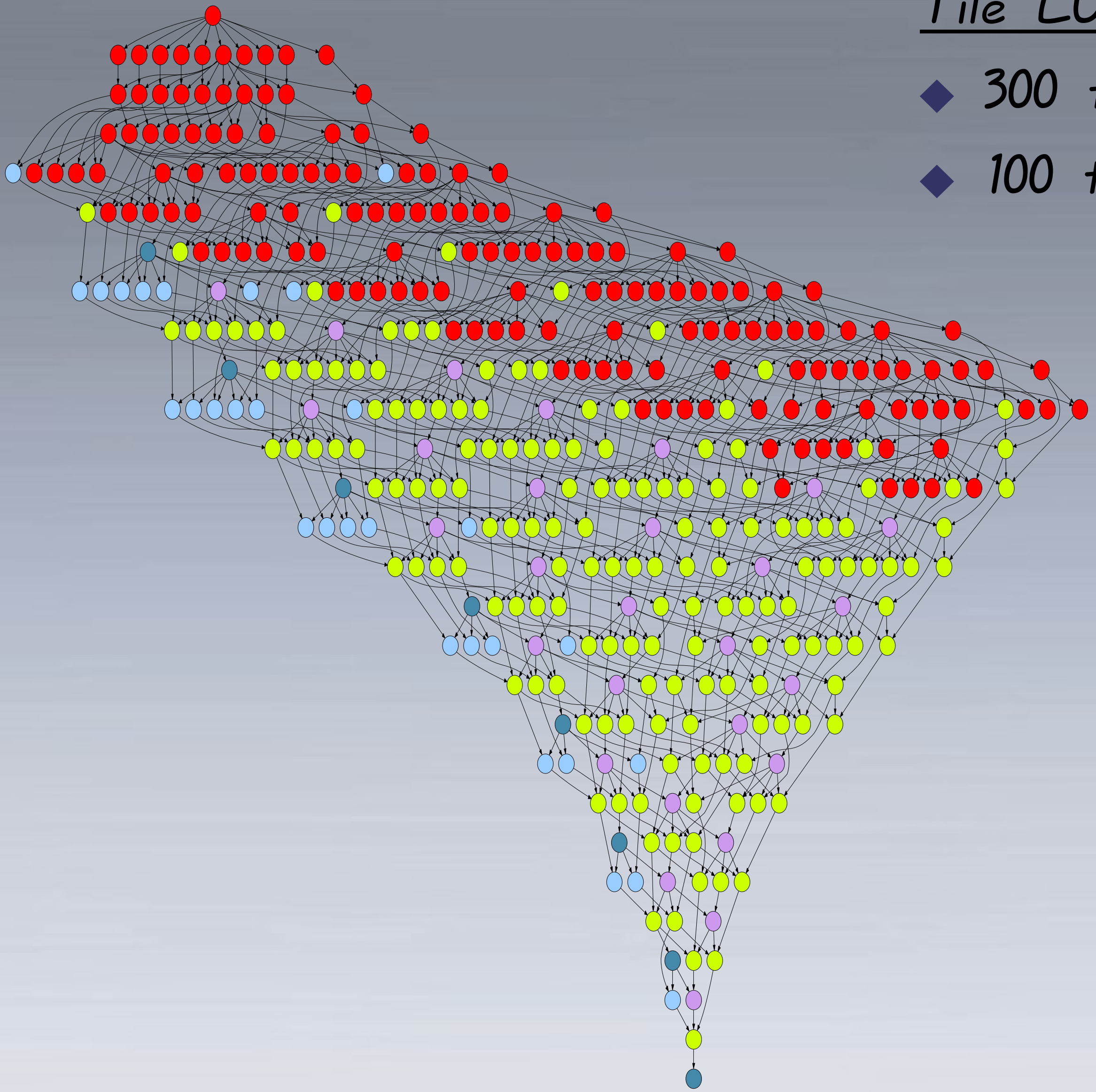
Exploring the DAG by a Sliding Window



Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window

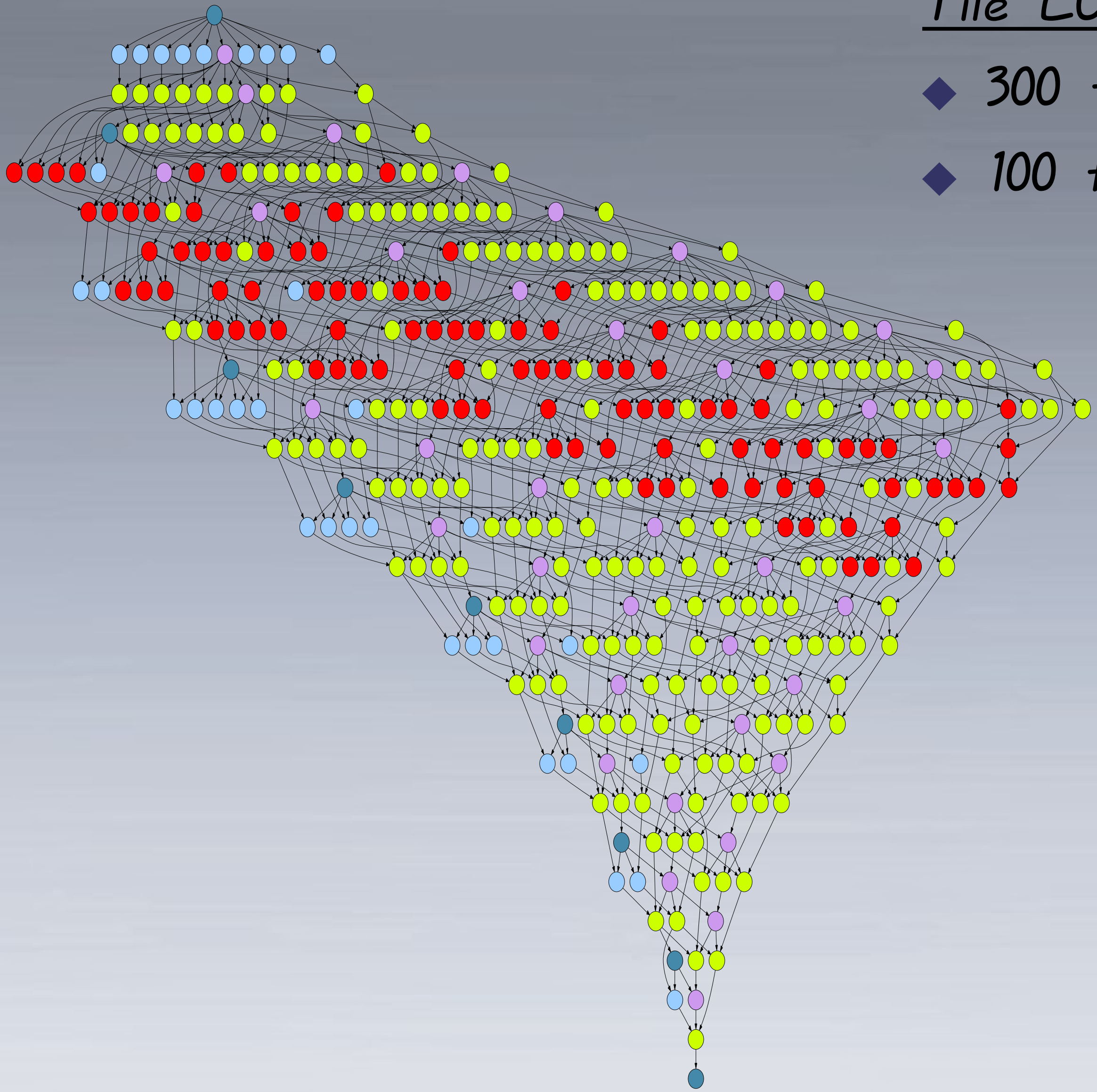
Exploring the DAG by a Sliding Window



Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window

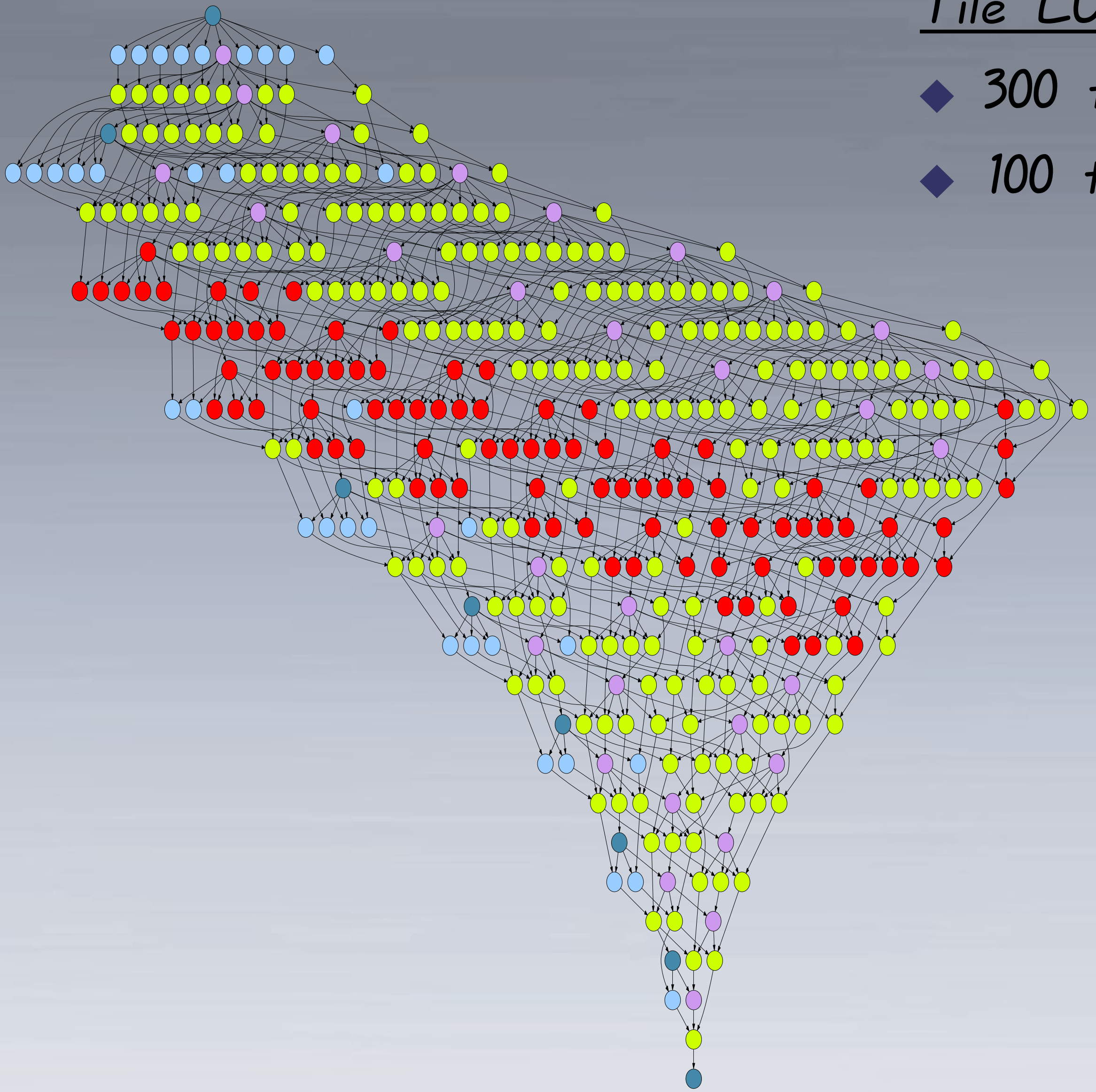
Exploring the DAG by a Sliding Window



Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window

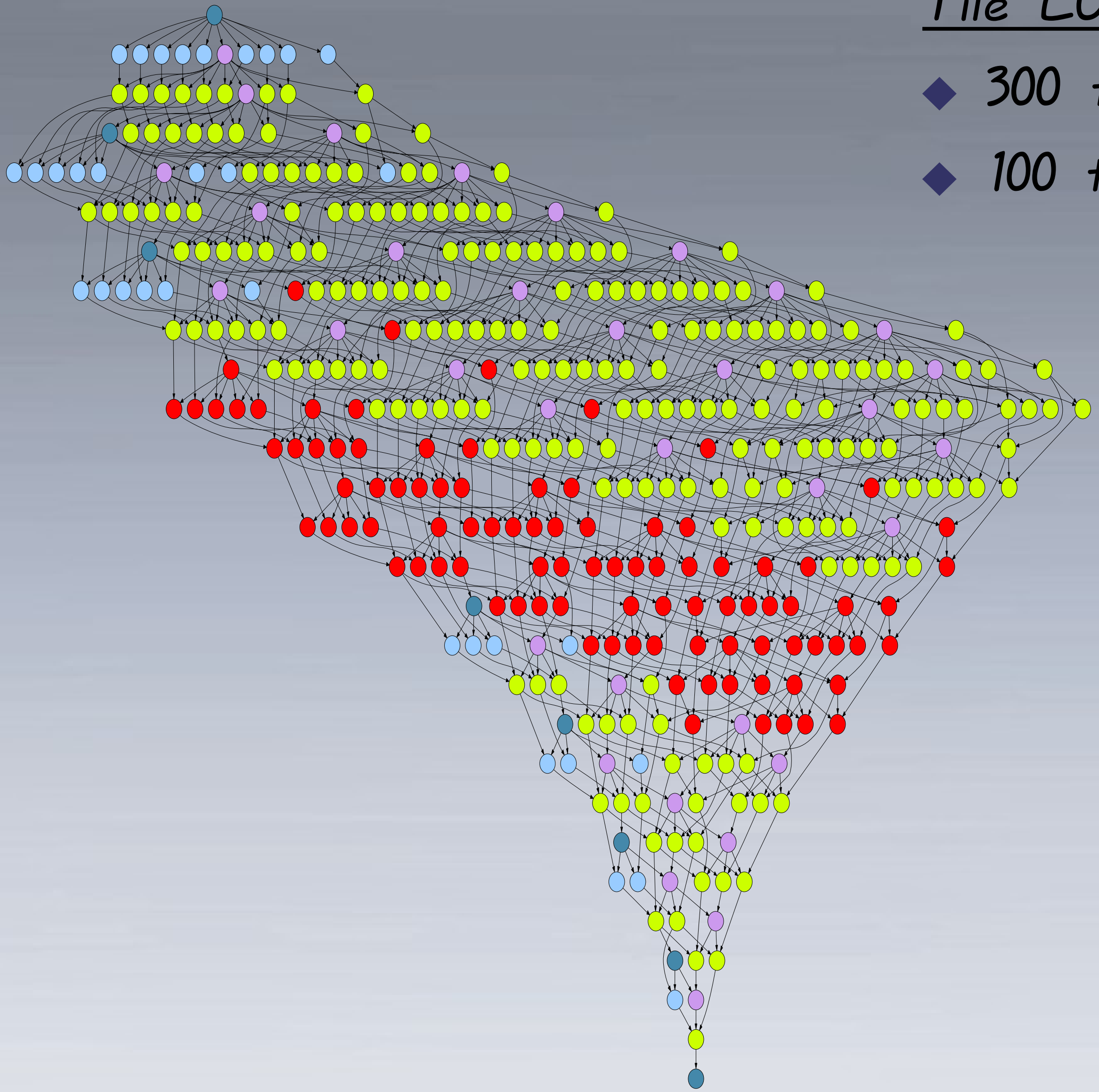
Exploring the DAG by a Sliding Window



Tile LU factorization 10x10 tiles

- *300 tasks total*
- *100 task window*

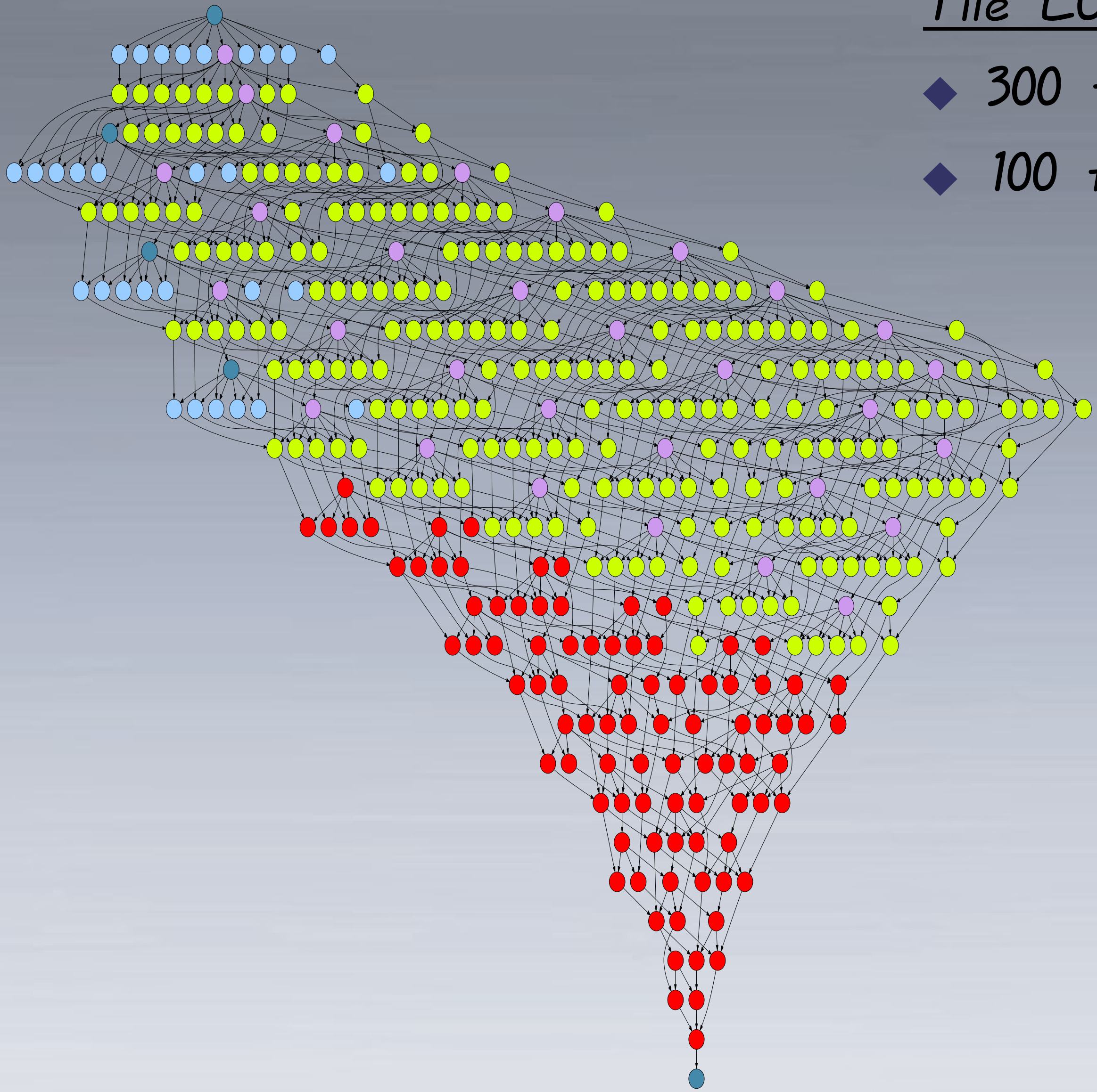
Exploring the DAG by a Sliding Window



Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window

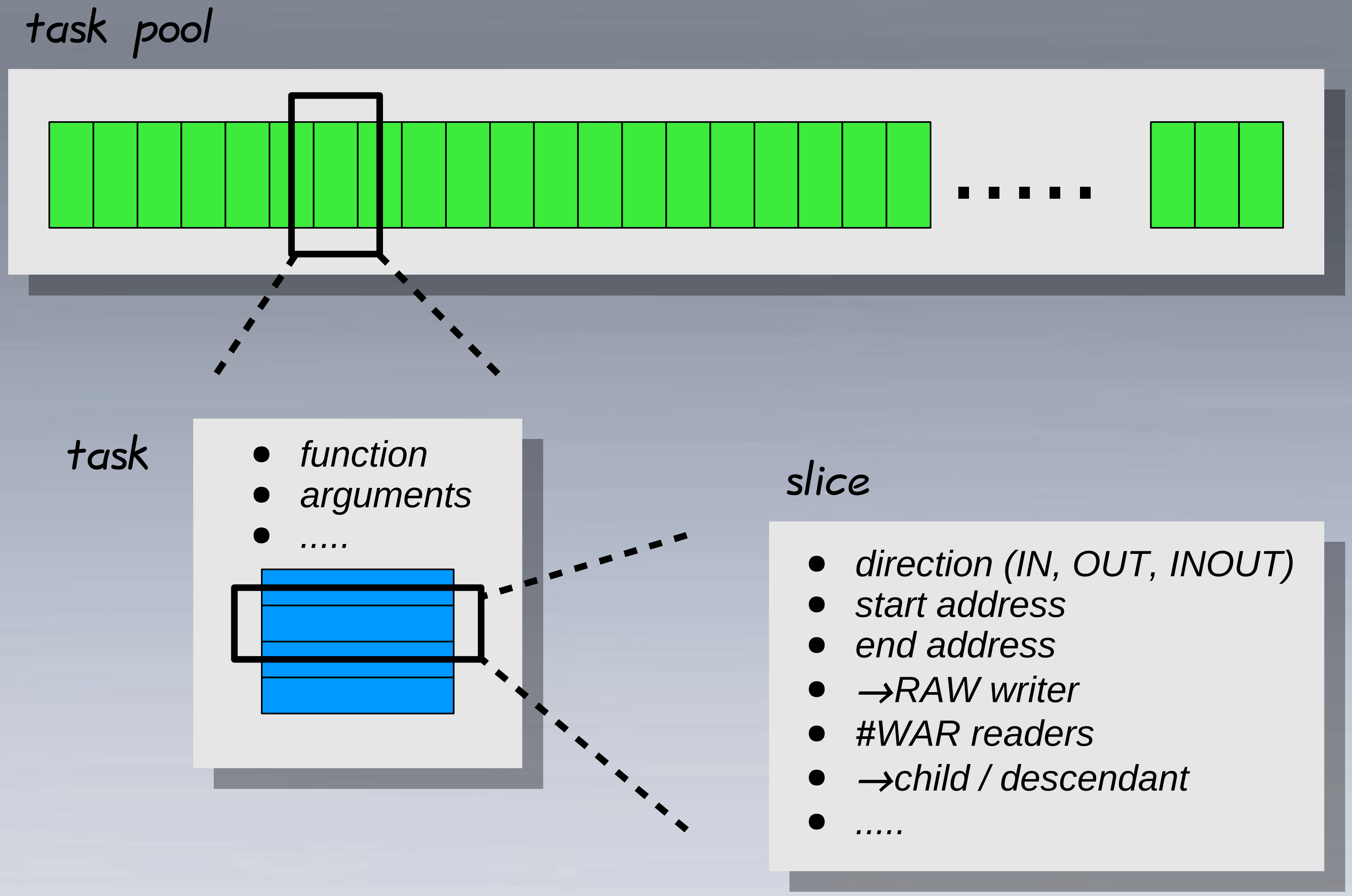
Exploring the DAG by a Sliding Window



Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window

GUST Organization



task pool

task

- *function*
- *arguments*
- *.....*

slice

- *direction (IN, OUT, INOUT)*
- *start address*
- *end address*
- *→RAW writer*
- *#WAR readers*
- *→child / descendant*
- *.....*

- task – a unit of scheduling (quantum of work)
- slice – a unit of dependency resolution (quantum of data)

GUST Current State

Absent features

- WAW hazard not supported
  - *extremely unlikely to occur in dense linear algebra*
- no renaming for WAR hazard
  - *unlikely to provide benefits in dense linear algebra*
- Prioritizing of tasks
  - *easy to implement, but no compelling case so far*

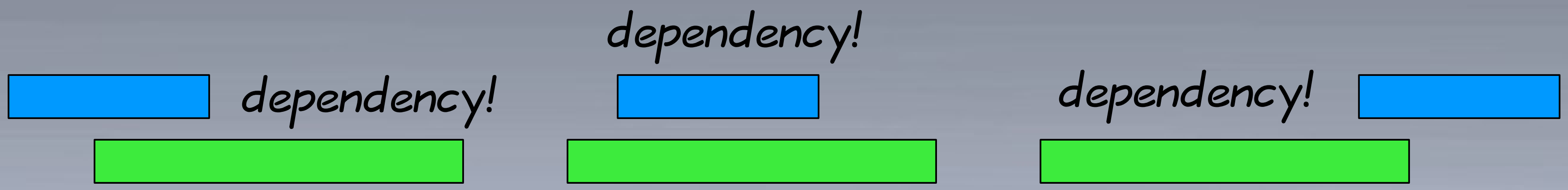
Lagging features

- One core devoted to queueing
  - *queueing requires optimizations*

GUST Current State

Extra features

- partially overlapping memory regions



- Ease of dropping dependencies with the NODEP parameter
- Prioritizing of data paths (to be implemented shortly)

Bottom Line - good performance
comparable to SMPSs
occasionally better
not too far from the static schedule

Dropping a Dependency

```
Insert_Task(SCHED_dgessm,
    &NB       , sizeof(int)         , VALUE,
    &NB       , sizeof(int)         , VALUE,
    &NB       , sizeof(int)         , VALUE,
    &IB       , sizeof(int)         , VALUE,
    IPIV(k, k), NB*sizeof(double)    , INPUT,
    A(k, k)   , NB*NB*sizeof(double), NODEP,
    &NB       , sizeof(int)         , VALUE,
    A(k, n)   , NB*NB*sizeof(double), INOUT,
    &NB       , sizeof(int)         , VALUE,
    NULL);
```

- Allows to easily drop dependency check on a parameter.
- Allows for fine tuning the schedule in certain cases
- Proved necessary in implementing the tile QR algorithm.

Prioritizing a Data Path

Tile LU – parallel

```
for (k = 0; k < BB; k++) {
    Insert_Task(CORE_dgetrf,
        A(k, k), NB*NB*sizeof(double), INOUT,
        IPIV(k, k), NB*sizeof(double), OUTPUT,

    for (n = k+1; n < BB; n++)
        Insert_Task(CORE_dgessm,
            IPIV(k, k), NB*sizeof(double), INPUT,
            A(k, k), NB*NB*sizeof(double), NODEP,
            A(k, n), NB*NB*sizeof(double), INOUT,

    for (m = k+1; m < BB; m++) {
        Insert_Task(CORE_dtstrf,
            A(k, k), NB*NB*sizeof(double), INOUT | PRIORITY,
            A(m, k), NB*NB*sizeof(double), INOUT,
            L(m, k), NB*IB*sizeof(double), OUTPUT,
            IPIV(m, k), NB*sizeof(double), OUTPUT,

        for (m = k+1; m < BB; m++)
            Insert_Task(CORE_dssssm,
                A(k, n), NB*NB*sizeof(double), INOUT | PRIORITY,
                A(m, n), NB*NB*sizeof(double), INOUT,
                L(m, k), NB*IB*sizeof(double), INPUT,
                A(m, k), NB*NB*sizeof(double), INPUT,
                IPIV(m, k), NB*sizeof(double), INPUT,
    }
}
```

- easy to implement
- more powerful than task prioritization

Can potentially close the gap between the dynamic schedule and the static schedule.

Future .....

Work in Progress . . . . .

silly picture from Internet here . . .