DAGuE: A Generic Distributed DAG Engine for HPC
Hardware Complexity
- Hierarchies of Multi-Cores
- Non Uniform Memory Access
- Accelerators
- Networks with deep hierarchies

Portability
- Programming Portability
- Performance Portability

Calls for Dynamic / Asynchronous Programming Model
DAGuE Goals

• Keep the algorithm as simple as possible
  • Depict only the flow of data between tasks
  • *Distributed Dataflow Environment based on Dynamic Scheduling of (Micro) Tasks*

• Programmability: layered approach
  • Algorithm / Data Distribution

• Portability / Efficiency
  • Use all available hardware; overlap comm / comp

• Decouple “System issues” from Algorithm
DAGuE toolchain

User

Input Serial Code
Imperfectly Nested
Affine Loops

DAGuE Compiler

Symbolic
Representation of
the DAG

JDF Translator

Parallel Tasks
Stubs

System Compiler

User Code:
Kernel Bodies &
main program

DAGuE Compiler

DAGuE runtime
Library

MPI / Pthreads
Parallel
Program

Omega Test

DAGuE Compilers and Runtime
Example: Cholesky Factorization

- **Cholesky Decomposition**
  - Let $A$ be a real symmetric positive definite matrix
  - Find $L$ such that $A = LL^T$

Tiled Algorithm in A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Computing, 2008
Input Format: SMPSS-Like

FOR \( k = 0 \ldots \text{TILES}-1 \)
\[
A[k][k] \leftarrow \text{DPOTRF}(A[k][k])
\]

FOR \( m = k+1 \ldots \text{TILES}-1 \)
\[
A[m][k] \leftarrow \text{DTRSM}(A[k][k], A[m][k])
\]

FOR \( n = k+1 \ldots \text{TILES}-1 \)
\[
A[n][n] \leftarrow \text{DSYRK}(A[n][k], A[n][n])
\]

FOR \( m = n+1 \ldots \text{TILES}-1 \)
\[
A[m][n] \leftarrow \text{DGEMM}(A[m][k], A[n][k], A[m][n])
\]
Input Format: Job Data Flow

TRSM(k, n)
// Execution space
k = 0..SIZE-1
n = k+1..SIZE-1
: A(n, k) // Parallel Partitionning
READ  T <- T POTRF(k)
RW    C <- (k == 0) ? A(n, k)
    : C GEMM(k-1, n, k)
    -> A SYRK(k, n)
    -> A GEMM(k, n+1..SIZE-1, n)
    -> B GEMM(k, n, k+1..n-1)
    -> A(n, k)
From Seq. to JDF

- DAGuE Compiler
  - Analysis the data flow using algebraic expressions
  - Omega Test used to compute algebraic relations between edges
  - Imperfectly nested affine loop tests
  - Anti-Dependencies may introduce additional control edges
Runtime DAG Representation

- Every process has the algebraic DAG rep.
- Dist. Scheduling based on remote completion notifications
- NUMA / Cache aware Scheduling
- Work Stealing and sharing based on memory hierarchies
Runtime DAGuE Engine

- Data Distribution (and data/task affinity) imposes a task location
- On each node, the full DAG algebraic representation is available
- Each computing unit (core, GPU, etc.) runs its own instance of the DAGuE scheduler
- An additional communication thread sends completion notifications and data when necessary
Scheduling in DAGuE

- Based on Work Stealing
  - Shared data structures with atomic access operations
  - Uniform scheduler: all scheduler run with the global view of the DAG and the local view of progress (plus remote notifications)
  - Fully Distributed scheduler: all threads alternate between scheduling and work
- Main heuristic: data locality
  - DAGuE engine tracks data usage, and targets to improve data reuse
  - NUMA aware hierarchical bounded buffers to implement work stealing
- Users hints: tasks with “high priority”; Algebraic expressions for priorities
  - Insertion in waiting queue abides to priority, but work stealing can alter this ordering
- Communications heuristics
  - Communications inherits priority of destination task
Example: RTT

Sequential code:

```c
B = ping(A[0]);
for(k = 1; k < NT-1; k++) {
    C = pong(B);
    B = ping(C);
}
A[0] = B;
```

- Read data into B using ping
- Call pong on B to make C
- Call ping on C to make B
- Iterate NT-1 times
- Save data
Example RTT: JDF

- The JDF Translator will create a stub (rtt.c / rtt.h) which includes two visible functions:
  - `rtt_new(A, NT)` creates a DAG generator for a specific data A
  - Data format includes accessors to discover the data distribution
  - `rtt_destroy()` frees the resources allocated by the new 

```c
1   PING(k)
2   k = 0 .. NT          // Execution space
3       : A[0]          // Parallel partitioning
4   T ← (k == 0) ? A(0) : I PONG(k-1) [ATYPE]
5       → (k == NT) ? A(0) : I PONG(k) [ATYPE]

6   PONG(k)
7   k = 0 .. NT-1       // Execution space
9   I ← T PING(k)       [ATYPE]
10       → T PING(k+1) [ATYPE]
11
```
Example RTT: main program

```c
#include "ping_pong.h"
int main( int argc, char** argv) {
  int size, rank; /* MPI_COMM_WORLD size and rank */
  dague_object_t* rtt;
  dague_matrix_t* A = dague_matrix( 1000, [ATYPE],
                                   "cyclic", 1, size );

  dague_t* mydag = dague_init("-c 1");
  rtt = rtt_new( A, rank, 1000);
  dague_enqueue( my_dag, rtt );
  dague_progress( my_dag );
  dague_fini( &my_dag );
}
```
Example: Reduction Operation

- Apply a user defined operator on each data and store the result in a single location.
- Suppose the operator is associative and commutative.
Example: Reduction Operation

- Apply a user defined operator on each data and store the result in a single location.
- Suppose the operator is associative and commutative.
Example: Reduction Operation

```plaintext
0 reduce(l, p)
   l = 1 .. depth+1
   p = 0 .. (MT / (1<<l))
   : A( p )
   READ A <- (1 == l) ? A(2*p) : C reduce( l - 1, 2 * p )

1 READ B <- ((p * (1 << l) + (1 << (l-1))) > MT) ? A(0)
   <- (1 == l) ? A(2*p+1)
   <- (1 != l) ? C reduce(l - 1, p * 2 + 1)

2 WRITE C -> ((depth+1) == l) ? R(p)
   -> (0 == (p%2)) ? A reduce(l+1, p/2)
       :B reduce(l+1, p/2)
```
DAGuE: Analysis Tools

Hermitian Band Diagonal; 16x16 tiles
Experimental Platform

Dancer @ UTK

- 32 Cores (8 sockets)
- Intel Q9400 quad cores @ 2.5GHz
- 4GB RAM
- 2x 1GB/s ethernet
- 4 nodes with Fermi GPU
- 4 nodes with Tesla GPU

MKL-10.1.0.015 / gcc 4.4 / gfortran 4.4
**Bulge Chasing in DAGuE**

```
zhbrdt(sl,s,i)
/* Execution space */
sl = 0..NT-2
s = 0..NB-1
i = sl..NT-2

: A(0,i)
/* A == data_A(0,i) */
/* B == data_A(0,i+1) */

RW A <= (i==sl) & (0==s) & (i==i) ? A(0,i)
  <= (i==sl) & (i==s) & (i>=1) ? A zhbrdt(sl-1, NB-1, sl)
  <= (i==s) & (s>=1)          ? A zhbrdt(sl, s-1, s)
  <= (s1<i)                  ? B zhbrdt(sl, s, i-1)

-> ((1+s)==NB) & (i==sl)    ? A(0,i)
-> ((1+s)==NB) & ((1+s)<=i) ? A zhbrdt(s1+1, 0, s1+1)
-> ((1+s)<NB) & ((1+s)<=i)  ? A zhbrdt(s1, s+1, i-1)

RW B <= (0==s) & (0==sl)    ? A in_dat(i+1)
  <= (0==s) & (NT-2==i)     ? A zhbrdt_half(sl-1, NB-1)
  <= (0==s)                ? A zhbrdt(sl-1, NB-1, i+1)
  <= ((NT-2)==i)           ? A zhbrdt_half(sl, s-1)
  <=                        A zhbrdt(sl, s-1, i+1)

-> ((NT-2)==i)             ? A zhbrdt_half(sl, s)
->                        A zhbrdt(sl, s, i+1)

zhbrdt_half(sl,s)
/* Execution space */
sl = 0..NT-1
s = 0..NB-1

: A(0,NT-1)
/* A == data_A(0,i) */

RW A <= ((NT-1)!==s1)     ? B zhbrdt(sl, s, NT-2)
  <= (0==s)                ? A zhbrdt_half(sl-1, NB-1)

-> ((NT-1)==s1)           ? A zhbrdt_half(sl+1, NB-1)
-> ((NT-1)==s)            ? A zhbrdt_half(sl+1, 0)
-> ((NB-1)==s)            ? B zhbrdt(sl+1, 0, NT-2)

-> ((1+s)==NB) & (i==s1)  ? A(0,i)
-> ((1+s)==NB) & ((1+s)<s1) ? A zhbrdt(sl+1, 0, s1)
-> ((1+s)<NB) & ((1+s)<s1) ? A zhbrdt(s1, s+1, i)

-> ((NT-1)==s1)          ? A zhbrdt_half(sl+1, s)
-> ((NT-1)==s)           ? A zhbrdt_half(sl+1, i+1)
-> ((NB-1)==s)           ? B zhbrdt(sl+1, s+1, NT-2)
```

---

```c
extern "C" {

plasma.h

#include <plasma.h>
#include <core_blas.h>
#include "dague.h"
#include "data_distribution.h"
#include "data_dist/matrix/matrix.h"
#include "dplasma/core/generated/core_z.h"
#include "dplasmajdf.h"

A          [type = "tiled_matrix_desc_t*"
NT         [type = int
NB         [type = int

in_dat(i) [profile = off hidden = on
i = 1..(NT-1)
: A(0,i)

A           [type = "tiled_matrix_desc_t*"
```

---

```c
/* A == data_A(0,i) */
/* B == data_A(0,i+1) */

/* nothing */
printlog("thread %d in_dat(%d, %d, %d) <
	A[0, %d] = %p",
context > eu_id, 0, 0, i, i, A);

/* B == data_A(0,i+1) */

/* A == data_A(0,i) */
/* B == data_A(0,i+1) */

/* nothing */
printlog("thread %d CORE_zhbrdt_half(s1 = %d, s = %d)
	(A(%d,%d)[%p], s1*NB+s, NT-1)
", context > eu_id, s1, s, 0, NT-1, A);

/* nothing */
printlog("thread %d CORE_zhbrdt(s1 = %d, s = %d, i = %d)
	(A(%d,%d)[%p], A(%d,%d)[%p], s1*NB+s, i)
", context > eu_id, s1, s, i, 0, i, A, 0, i+1, B);
```
Bulge Chasing in DAGuE

- s1 = 0; s = 0
- s1 = 0; s = 1
- s1 = 0; s = 2
- s1 = 0; s = 3
- s1 = 1; s = 0
- s1 = 1; s = 1
- s1 = 1; s = 2
- s1 = 1; s = 3
- s1 = 2; s = 0
- s1 = 2; s = 1
- s1 = 2; s = 2
- s1 = 2; s = 3
- s1 = 3; s = 0
- s1 = 3; s = 1
- s1 = 3; s = 2
- s1 = 3; s = 3
- s1 = 4; s = 0
- s1 = 4; s = 1
- s1 = 4; s = 2
- s1 = 4; s = 3
Conclusion

- Hybrid programming (of dense LA) made easy(ier)
  - Portability: inherently take advantage of all hardware capabilities
  - Efficiency: deliver the best performance on tested algorithms
- Works well with Dense Linear Algebra with Direct Method
  - Sparse?
  - Branch and Bound?
  - Iterative Method?
- Let different people focus on different problems
  - Application developers on their algorithms
  - System developers on system issues
Related Works

- YML Like: read the DAG representation; unroll it (completely); schedule it (centrally)
- PLASMA / MAGMA / T-BLAS / STARPU like: seq. code -> new tasks + dep.; tasks window; sched. is dyn/stat
- PTG like: use a concise DAG representation to discover new tasks. Have a bounded ready list.

DAGuE uses a concise representation of the DAG, instantiates dynamically the tasks, scheduling is fully distributed and dynamic