Where we are today

- Today software developers face systems with
  - ~1 TFLOP of compute power per node
  - 32+ of cores, 100+ hardware threads
  - **Highly heterogeneous** architectures (cores + specialized cores + accelerators/coprocessors)
  - Deep **memory hierarchies**
  - But wait there is more: They are distributed!
  - Consequence is **systemic load imbalance**
- And we still ask the same question: How to harness these devices productively?
  - SPMD produces choke points, wasted wait times
  - We need to improve efficiency, power and reliability
The missing parallelism

- Too difficult to find parallelism, to debug, maintain and get good performance for everyone
- Increasing gaps between the capabilities of today’s programming environments, the requirements of emerging applications, and the challenges of future parallel architectures
Decouple “System issues” from Algorithm

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

• Programmability: layered approach
  • Algorithm / Data Distribution
  • Parallel applications without parallel programming

• Portability / Efficiency
  • Use all available hardware; overlap data movements / computation
  • Progress is still possible when imbalance arise
Dataflow with Runtime scheduling

• Algorithms need help to unleash their power
  • *Hardware specificities*: a runtime can provide portability, performance, scheduling heuristics, heterogeneity management, data movement, ...
  • *Scalability*: maximize parallelism extraction, but no centralized scheduling or entire DAG unpacking: dynamic and independent discovery of the relevant portions during the execution
  • *Jitter resilience*: Do not support explicit communications, instead make them implicit and schedule to maximize overlap and load balance

• The need to express the algorithms differently
Divide-and-orchestrate
Divide-and-orchestrate

- Leave the optimization of each level to specialized entities
  - Compiler do marvels when the cost of memory accesses is known
- Think hierarchical super-scalar
  - Compiler focus on problems that fit in the cache
  - Humans focus on depicted the flow of data between tasks
  - And a runtime orchestrate the flows to maximize the throughput of the architecture

Diagram:
- ScaLAPACK QR
- PLASMA QR
- GEQRT
- TSQRT
- UNMQR
- TSMQR
The PaRSEC framework

Cores
Memory Hierarchies
Coherence
Data Movement

Parallel Runtime
Scheduling
Data
Data Movement
Tasks

Domain Specific Extensions
Compact Representation - PTG
Dynamic Discovered Representation - DTG

Specialized Kernels

Hardware
Cores
Accelerators
Domain Specific Extensions

• DSEs ⇒ higher productivity for developers
  • High-level data types & ops tailored to domain
  • E.g., relations, matrices, triangles, …

• Portable and scalable specification of parallelism
  • Automatically adjust data structures, mapping, and scheduling as systems scale up
  • Toolkit of classical data distributions, etc
PaRSEC Compiler

Serial Code to Dataflow Representation
Example: QR Factorization

FOR $k = 0 .. \text{SIZE} - 1$

$A[k][k], T[k][k] \leftarrow \text{GEQRT}( A[k][k] )$

FOR $m = k+1 .. \text{SIZE} - 1$

$A[k][k]|\text{Up}, A[m][k], T[m][k] \leftarrow \text{TSQRT}( A[k][k]|\text{Up}, A[m][k], T[m][k] )$

FOR $n = k+1 .. \text{SIZE} - 1$

$A[k][n] \leftarrow \text{UNMQR}( A[k][k]|\text{Low}, T[k][k], A[k][n] )$

FOR $m = k+1 .. \text{SIZE} - 1$

$A[k][n], A[m][n] \leftarrow \text{TSMQR}( A[m][k], T[m][k], A[k][n], A[m][n] )$
for (k = 0; k < A.mt; k++) {
    Insert_Task( zgeqrt, A[k][k], INOUT, T[k][k], OUTPUT);
    for (m = k+1; m < A.mt; m++) {
        Insert_Task( ztsqrt, A[k][k], INOUT | REGION_D | REGION_U,
                    A[m][k], INOUT | LOCALITY, T[m][k], OUTPUT);
    }
    for (n = k+1; n < A.nt; n++) {
        Insert_Task( zunmqr, A[k][k], INPUT | REGION_L,
                    T[k][k], INPUT,
                    A[k][m], INOUT);
        for (m = k+1; m < A.mt; m++)
            Insert_Task( ztsmqr, A[k][n], INOUT,
                        A[m][n], INOUT | LOCALITY,
                        A[m][k], INPUT,
                        T[m][k], INPUT);
    }
}
Dataflow Analysis

- data flow analysis
  - Example on task DGEQRT of QR
  - Polyhedral Analysis through Omega Test
  - Compute algebraic expressions for:
    - Source and destination tasks
    - Necessary conditions for that data flow to exist

FOR \( k = 0 \) .. SIZE - 1
\[
A[k][k], T[k][k] \leftarrow \text{GEQRT}(A[k][k])
\]

FOR \( m = k+1 \) .. SIZE - 1
\[
A[k][k]|_{\text{Up}}, A[m][k], T[m][k] \leftarrow \text{TSQRT}(A[k][k]|_{\text{Up}}, A[m][k], T[m][k])
\]

FOR \( n = k+1 \) .. SIZE - 1
\[
A[k][n] \leftarrow \text{UNMQR}(A[k][k]|_{\text{Low}}, T[k][k], A[k][n])
\]

FOR \( m = k+1 \) .. SIZE - 1
\[
A[k][n], A[m][n] \leftarrow \text{TSMQR}(A[m][k], T[m][k], A[k][n], A[m][n])
\]
Intermediate Representation: Job Data Flow

GEQRT(k)
/* Execution space */
k = 0..( MT < NT ) ? MT-1 : NT-1 )
/* Locality */
: A(k, k)
RW A <- (k == 0)  ? A(k, k)
    : A1 TSMQR(k-1, k, k)
    -> (k < NT-1)  ? A UNMQR(k, k+1 .. NT-1)  [type = LOWER]
    -> (k < MT-1)  ? A1 TSQRT(k, k+1)       [type = UPPER]
    -> (k == MT-1) ? A(k, k)                 [type = UPPER]
WRITE T <- T(k, k)
    -> T(k, k)
    -> (k < NT-1) ? T UNMQR(k, k+1 .. NT-1)
/* Priority */
:(NT-k)*(NT-k)*(NT-k)

BODY
zgeqrt( A, T )
END

Control flow is eliminated, therefore maximum parallelism is possible
Dataflow Representation

PaRSEC Compiler

Additional libraries:
- MPI
- CUDA
- pthreads
- PLASMA
- MAGMA

Application code & Codelets

Supercomputer
Example: Reduction Operation

- Reduction: apply a user defined operator on each data and store the result in a single location.
  
  (Suppose the operator is associative and commutative)

```c
for(s = 1; s < N/2; s = 2*s)
    for(i = 0; i < N-s; i += s)
        operator(V[i], V[i+s])
```

**Issue:** Non-affine loops lead to non-polyhedral array accessing
Example: Reduction Operation

\[
\text{reduce}(l, p) : V(p) \\
l = 1 \ldots \text{depth+1} \\
p = 0 \ldots (\text{MT} / (1 \ll l))
\]

\[
\text{RW} \quad A \leftarrow (1 = l) \Rightarrow V(2*p) \\
\text{} \quad : A \text{ reduce}( l-1, 2*p ) \\
\text{} \quad \Rightarrow ((\text{depth+1}) = l) \Rightarrow V(0) \\
\text{} \quad \Rightarrow (0 = (p\%2)) \Rightarrow A \text{ reduce}(l+1, p/2) \\
\text{} \quad \quad : B \text{ reduce}(l+1, p/2)
\]

\[
\text{READ} \quad B \leftarrow ((p*(1\ll l) + (1\ll(l-1))) > \text{MT}) \Rightarrow V(0) \\
\text{} \quad \leftarrow (1 = l) \Rightarrow V(2*p+1) \\
\text{} \quad \leftarrow (1 != l) \Rightarrow A \text{ reduce}( l-1, p*2+1 )
\]

\[
\text{BODY} \quad \text{operator}(A, B); \quad \text{END}
\]

**Solution:** Hand-writing of the data dependency using the intermediate Data Flow representation
Data/Task Distribution

- Flexible data distribution
  - Decoupled from the algorithm
  - Expressed as a user-defined function
  - Only limitation: must evaluate uniformly across all nodes

- Common distributions provided in DSEs
  - 1D cyclic, 2D cyclic, etc.
  - Symbol Matrix for sparse direct solvers
Algorithm is now expressed as a DAG (potentially Parameterized)

Parallel Runtime
• DAG too large to be generated ahead of time
  • Generate it dynamically
  • Merge parameterized DAGs with dynamically generated DAGs

HPC is about distributed heterogeneous resources
  • Have to get involved in message passing
  • Distributed management of the scheduling
  • Dynamically deal with heterogeneity
Runtime DAG scheduling

- Every process has the symbolic DAG representation
  - Only the (node local) frontier of the DAG is considered
  - Distributed Scheduling based on remote completion notifications
- Background remote data transfer automatic with overlap
- NUMA / Cache aware Scheduling
  - Work Stealing and sharing based on memory hierarchies
Scheduling Heuristics in PaRSEC

- Manages parallelism & locality
  - Achieve efficient execution (performance, power, …)
  - Handles specifics of HW system (hyper-threading, NUMA, …)
- Per-object capabilities
  - Read-only or write-only, output data, private, relaxed coherence
  - engine tracks data usage, and targets to improve data reuse
  - NUMA aware hierarchical bounded buffers to implement work stealing
- Users hints: expressions for distance to critical path
  - Insertion in waiting queue abides to priority, but work stealing can alter this ordering due to locality
- Communications heuristics
  - Communications inherits priority of destination task
- Algorithm defined scheduling
Now we have a runtime and some algorithms

What’s next?
Auto-tuning

- Multi-level tuning
  - Tune the kernels based on local architecture
  - Then tune the algorithm
- Depends on the network, type and number of cores
- For a fixed size matrix increasing the task duration (or the tile size) decrease parallelism
- For best performance: auto-tune per system

Weak-scaling

% efficiency

Block Size (NB)

1 Nodes (8 cores) 4 Nodes (32 cores) 81 Nodes (648 cores)

Not enough parallelism
References


81 dual Intel Xeon L5420@2.5GHz (2x4 cores/node) → 648 cores
MX 10Gbs, Intel MKL, Scalapack

81 dual Intel Xeon L5420@2.5GHz
(2x4 cores/node) → 648 cores
MX 10Gbs, Intel MKL, Scalapack
Scalability in Distributed Memory

- Parameterized Task Graph representation
- Independent distributed scheduling

Scalable
HPL or LU with Partial Pivoting

Inria Bordeaux, UTK

- Why LU is different? 2 reasons:
  - The panel is expressed using dataflow, generating many tiny tasks
    - Provides very poor performance
  - The DAG depends on the content of the data
    - The strict dataflow model we imposed on this exercise prevents message aggregation
    - Our implementation is not message optimal
  - Explore all cases with as less tasks as possible

Dancer: 16*8 cores E5520, IB 20Gbs, Intel MKL

Solid = 8 cores/node
Dashed = 7 cores/node
Scalapack
Partial pivoting

GEMM PEAK
Without pivoting
Incremental pivoting

Gigaflop/s
0 200 400 600 800 1000 1200
0 10000 20000 30000 40000 50000
Matrix Size

For that, we used the Netlib version. ScaLAPACK was run by assigning one process MPI per core.

We implemented Task flow for panel factorization and Task flow for swapping operations in update. We used a 2D block cyclic distribution for matrix. The size of tile used was 200 for DAGuE and 120 for ScaLAPACK and has been chosen to get the best asymptotic performance. The matrices were generated randomly with the same side for all measurements.

For each measurement, we do five iterations of the execution, we do not take in count the maximal and minimal values obtained. We display the average of the three other values. In most of the tests, we obtain a low standard deviation (less than 1 Gflop/s).

The results obtained are encouraging because our implementation outperform the ScaLAPACK implementation and reached 75% of the GEMM peak. However, the incremental pivoting algorithm which enables more parallelism and less synchronization in the panel factorization still provides better performance.
Hierarchical QR
ENS Lyon, Inria Bordeaux, UTK, UCD

- How to compose trees to get the best pipeline?
  - Flat, Binary, Fibonacci, Greedy, …
- Study on critical path lengths
- Surprisingly Flat trees are better for communications on square cases:
  - Less communications
  - Good pipeline
Sparse Direct Solvers
Inria Bordeaux, UTK

- Based on PaStiX solver
  - Super-nodal method as in SuperLU
  - Exploits an elimination tree => DAG
- LU, LLᵀ & LDLᵀ factorizations for shared memory with DAGuE/PaRSEC and StarPU
- GPU panel and update kernels
  - Based on blocked representation
- Problems:
  - Around 40 times more tasks than an equivalent dense factorization
  - Average task size can be very small (20x20)
  - Sizes are not regular
Heterogeneity Support

/* POTRF Lower case */
GEMM(k, m, n)

// Execution space
k = 0 .. MT-3  
m = k+2 .. MT-1  
n = k+1 .. m-1  

// Parallel partitioning
: A(m, n)

// Parameters
READ  A <- C TRSM(m, k)  
READ  B <- C TRSM(n, k)  
RW   C <- (k == 0) ? A(m, n) : C GEMM(k-1, m, n)  
     -> (n == k+1) ? C TRSM(m, n) : C GEMM(k+1, m, n)

BODY [CPU, CUDA, MIC, *]
• Multi GPU – single node
  - Single node
  - 4xTesla (C1060)
  - 16 cores (AMD opteron)

• Multi GPU - distributed
  - 12 nodes
  - 12xFermi (C2070)
  - 8 cores/node (Intel core2)
Energy efficiency

QR factorization (256 cores)

Total energy consumption

<table>
<thead>
<tr>
<th># Cores</th>
<th>Library</th>
<th>Cholesky</th>
<th>QR</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>ScaLAPACK</td>
<td>192000</td>
<td>672000</td>
</tr>
<tr>
<td></td>
<td>DPLASMA</td>
<td>128000</td>
<td>540000</td>
</tr>
<tr>
<td>256</td>
<td>ScaLAPACK</td>
<td>240000</td>
<td>816000</td>
</tr>
<tr>
<td></td>
<td>DPLASMA</td>
<td>96000</td>
<td>540000</td>
</tr>
<tr>
<td>512</td>
<td>ScaLAPACK</td>
<td>325000</td>
<td>1000000</td>
</tr>
<tr>
<td></td>
<td>DPLASMA</td>
<td>125000</td>
<td>576000</td>
</tr>
</tbody>
</table>

Work in progress with Hatem Ltaief

- Energy used depending on the number of cores
- Up to 62% more energy efficient while using a high performance tuned scheduling
  - Power efficient scheduling

SystemG: Virginia Tech Energy Monitored cluster (ib40g, intel, 8cores/node)
Analysis Tools

Hermitian Band Diagonal; 16x16 tiles
Conclusion

• Programming must be made easy(ier)
  • Portability: inherently take advantage of all hardware capabilities
  • Efficiency: deliver the best performance on several families of algorithms

• Computer scientists were spoiled by MPI
  • Now let’s think about our users

• Let different people focus on different problems
  • Application developers on their algorithms
  • System developers on system issues
  • Compilers on whatever they can
The end
Resilience

• The fault propagate in the system based on the data dependencies
• However, if the original data can be recovered, the execution complete without user interaction
• Automatic recovery made simple
Composition

- An algorithm is a series of operations with data dependencies
- A sequential composition limit the parallelism due to strict synchronizations
  - Following the flow of data we can loosen the synchronizations and transform them in data dependencies
Composition

- An algorithm is a series of operations with data dependencies
- A sequential composition limits the parallelism due to strict synchronizations
  - Following the flow of data, we can loosen the synchronizations and transform them into data dependencies
Composition

- An algorithm is a series of operations with data dependencies
- A sequential composition limits the parallelism due to strict synchronizations
  - Following the flow of data, we can loosen the synchronizations and transform them into data dependencies.
# Other Systems

<table>
<thead>
<tr>
<th></th>
<th>PaRSEC</th>
<th>SMPss</th>
<th>StarPU</th>
<th>Charm++</th>
<th>FLAME</th>
<th>QUARK</th>
<th>Tblas</th>
<th>PTG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
<td>Internal or Seq. w/ Affine Loops</td>
<td>Seq. w/ add_task</td>
<td>Seq. w/ add_task</td>
<td>Msg-Driver Objects</td>
<td>Internal (LA DSL)</td>
<td>Seq. w/ add_task</td>
<td>Seq. w/ add_task</td>
<td>Internal</td>
</tr>
<tr>
<td><strong>Accelerator</strong></td>
<td>GPU</td>
<td>GPU</td>
<td>GPU</td>
<td>GPU</td>
<td>GPU</td>
<td>GPU</td>
<td>GPU</td>
<td>GPU</td>
</tr>
<tr>
<td><strong>Availability</strong></td>
<td>Public</td>
<td>Public</td>
<td>Public</td>
<td>Public</td>
<td>Public</td>
<td>Public</td>
<td>Public</td>
<td>Not Avail.</td>
</tr>
</tbody>
</table>

Early stage: ParalleX
Non-academic: Swarm, MadLINQ, CnC

All projects support Distributed and Shared Memory (QUARK with QUARKd; FLAME with Elemental)
History: Beginnings of Data Flow

  - Coroutines, flow of data between process
- J.B. Dennis, 60’s
  - Data Flow representation of programs
  - Reasoning about parallelism, equivalence of programs, …
- “The semantics of a simple language for parallel programming”, G. Kahn
  - Kahn Networks