HPC: Linear Algebra Challenges

Arnaud Legrand, CNRS, University of Grenoble

LIG laboratory, arnaud.legrand@imag.fr

December 10, 2012

Reducing communication time

There are three main techniques for improving completion time:

- Tuning (overlap communication and computation)
- Ghosting (duplicate computation when there are dependencies)
- Scheduling (cache aware/cache oblivious, data distribution, ...)
- Change essence of the algorithm (e.g. Strassen $n^{2.80}$ or Winograd $n^{2.38}$) but this may be numerically harmful and beware of the O.

Outline

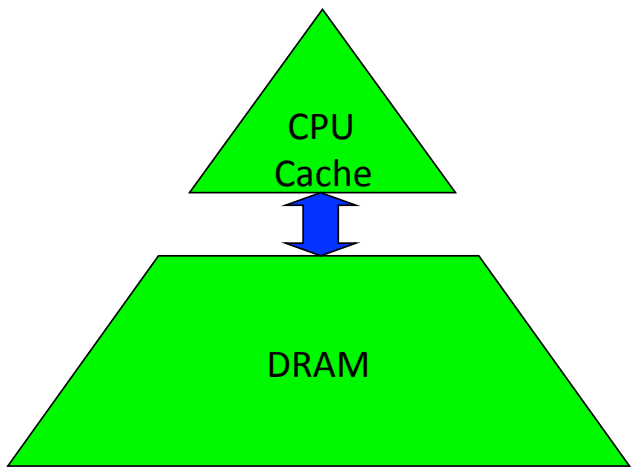
1. Communication "Avoiding" Algorithms
   - Cache oblivious algorithm
   - Parallel Algorithm
2. Synchronization-reducing algorithms
   - DAG generation
   - DAG generation
   - Granularity and Hybrid Computing
3. Auto-tuning
4. Reproducibility and Mixed-Precision Methods

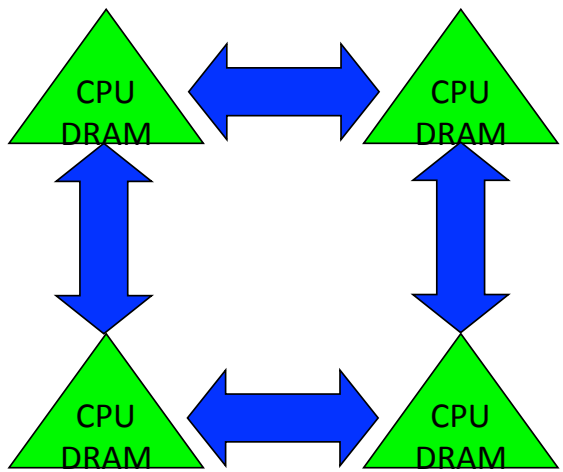
Outline

Why avoid communication? (1/2)

Algorithms have two costs (measured in time or energy):

1. Arithmetic (FLOPS)
2. Communication: moving data between
   - levels of a memory hierarchy (sequential case)
   - processors over a network (parallel case).





Courtesy of James Demmel

Why avoid communication? (2/3)

- Running time of an algorithm is sum of 3 terms:
  - # flops * time_per_flop
  - # words moved / bandwidth
  - # messages * latency

  } communication
- Time_per_flop << 1/ bandwidth << latency
  - Gaps growing exponentially with time [FOSC]

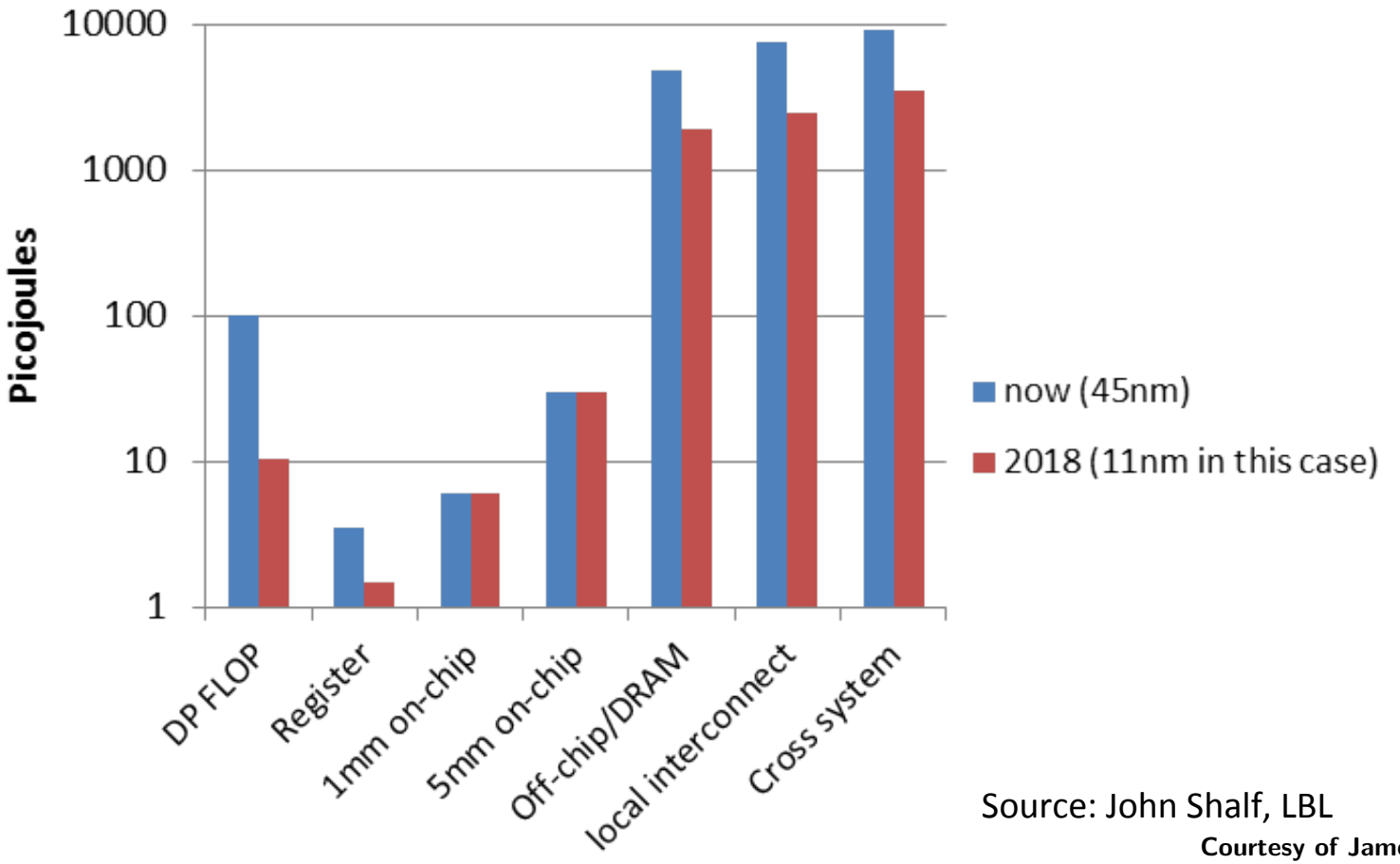
Annual improvements			
Time_per_flop		Bandwidth	Latency
59%	Network	26%	15%
	DRAM	23%	5%

- Avoid communication to save time

3

Courtesy of James Demmel

Why Minimize Communication? (2/2)

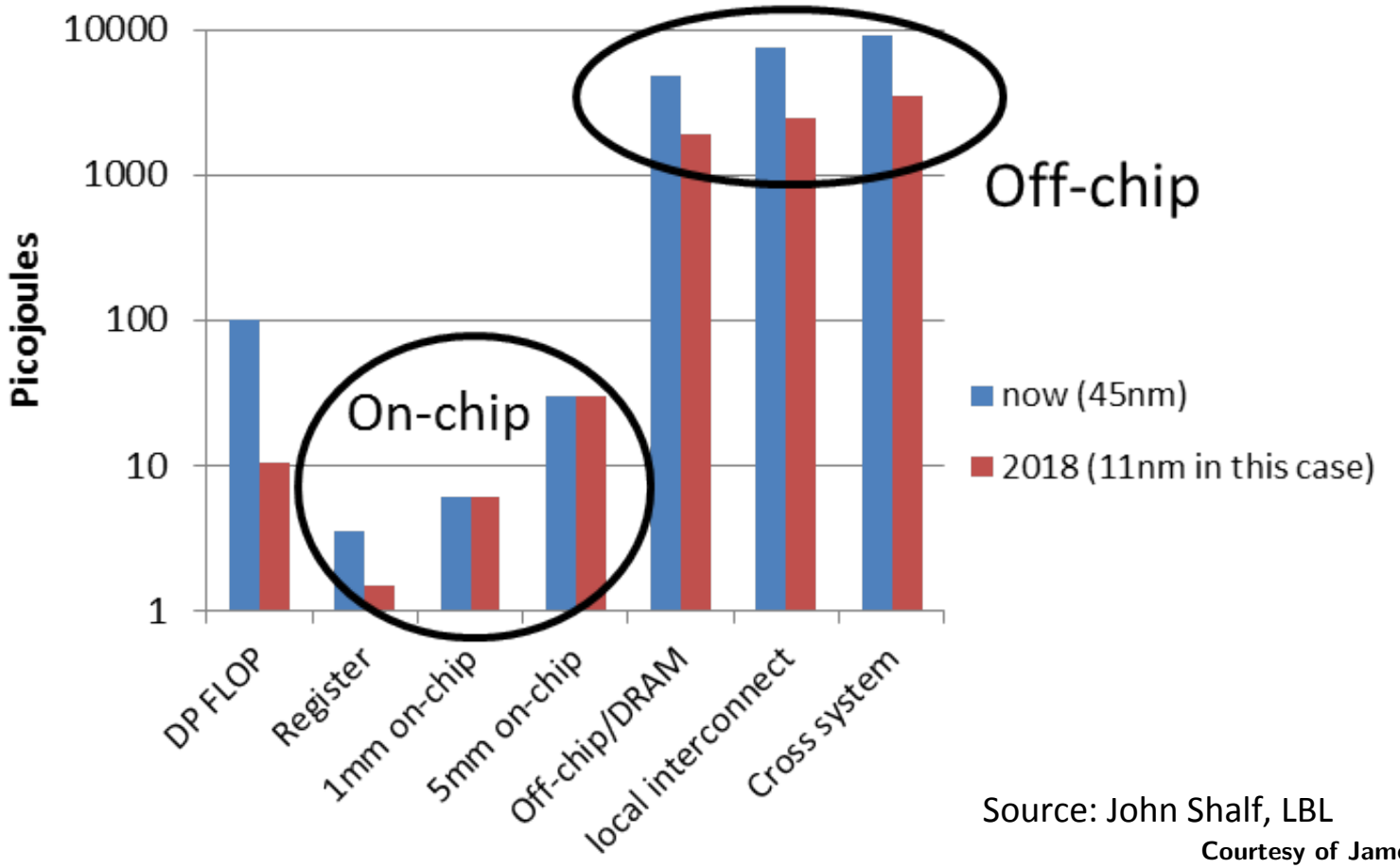


Source: John Shalf, LBL

Courtesy of James Demmel

Why Minimize Communication? (2/2)

Minimize communication to save energy



Source: John Shalf, LBL

Courtesy of James Demmel

Goals

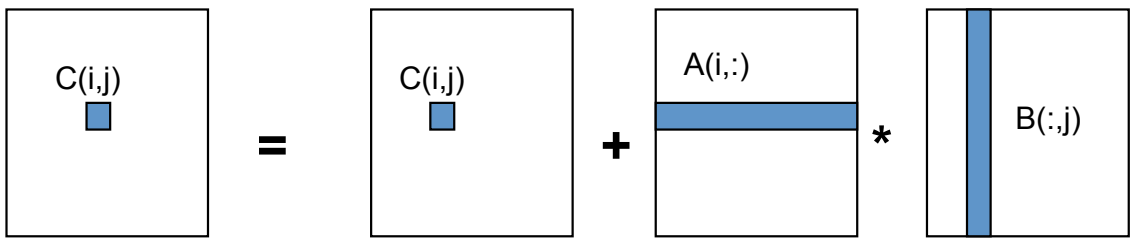
- Redesign algorithms to *avoid* communication
  - Between all memory hierarchy levels
    - L1 ↔ L2 ↔ DRAM ↔ network, etc
- Attain lower bounds if possible
  - Current algorithms often far from lower bounds
  - Large speedups and energy savings possible

Courtesy of James Demmel

Naïve Matrix Multiply

{implements C = C + A*B}

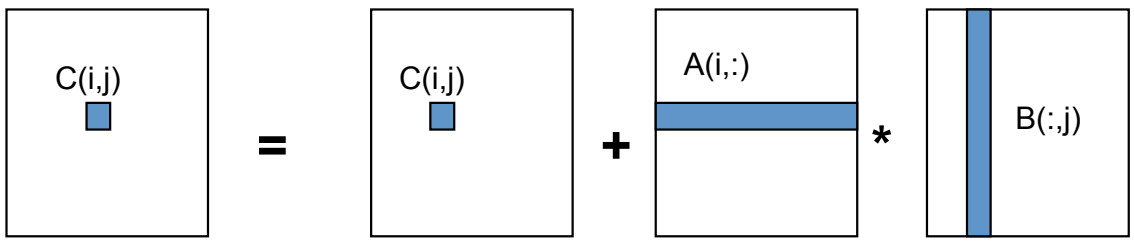
```
for i = 1 to n

  for j = 1 to n


    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
```



Courtesy of James Demmel

Naïve Matrix Multiply

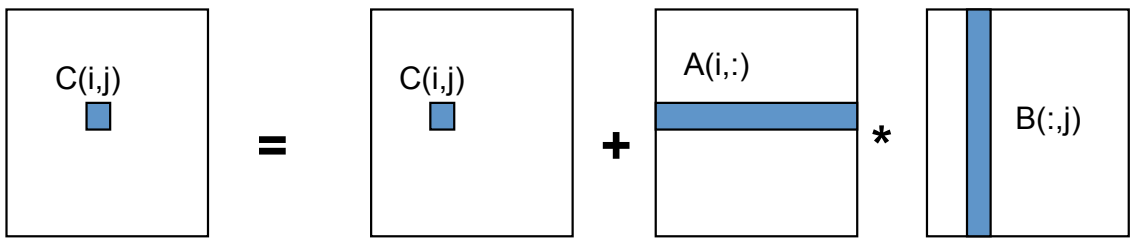
```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
  for j = 1 to n
     {read C(i,j) into fast memory}
     {read column j of B into fast memory}
     for k = 1 to n
        C(i,j) = C(i,j) + A(i,k) * B(k,j)
     {write C(i,j) back to slow memory}
```



Courtesy of James Demmel

Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}          ... n^2 reads altogether
   for j = 1 to n
      {read C(i,j) into fast memory}          ... n^2 reads altogether
      {read column j of B into fast memory}   ... n^3 reads altogether
      for k = 1 to n
         C(i,j) = C(i,j) + A(i,k) * B(k,j)
      {write C(i,j) back to slow memory}      ... n^2 writes altogether
```



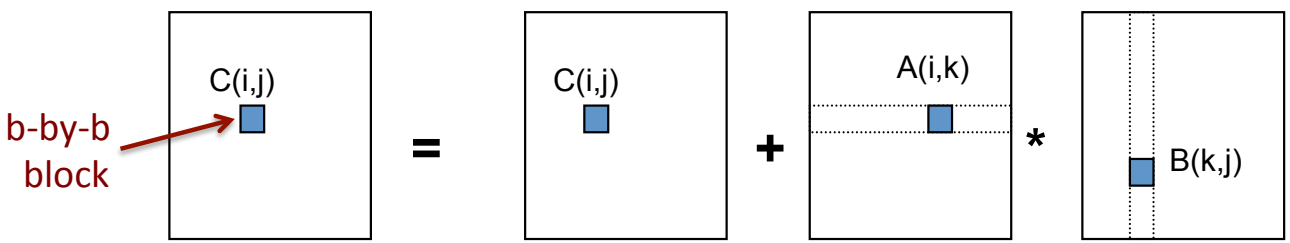
$n^3 + 3n^2$ reads/writes altogether – dominates $2n^3$ arithmetic

Courtesy of James Demmel

Blocked (Tiled) Matrix Multiply

Consider A,B,C to be n/b-by-n/b matrices of b-by-b subblocks where
b is called the block size; assume 3 b-by-b blocks fit in fast memory

```
for i = 1 to n/b
  for j = 1 to n/b
    {read block C(i,j) into fast memory}
    for k = 1 to n/b
      {read block A(i,k) into fast memory}
      {read block B(k,j) into fast memory}
      C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
    {write block C(i,j) back to slow memory}
```

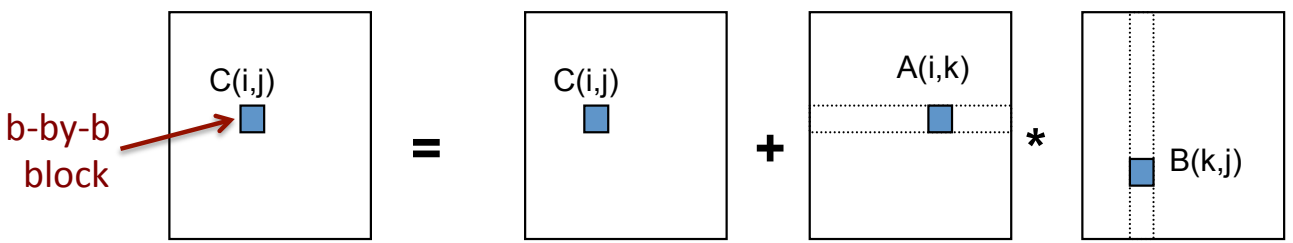


Courtesy of James Demmel

Blocked (Tiled) Matrix Multiply

Consider A,B,C to be n/b-by-n/b matrices of b-by-b subblocks where
b is called the block size; assume 3 b-by-b blocks fit in fast memory

```
for i = 1 to n/b
  for j = 1 to n/b
    {read block C(i,j) into fast memory}          ... b^2 × (n/b)^2 = n^2 reads
    for k = 1 to n/b
      {read block A(i,k) into fast memory}    ... b^2 × (n/b)^3 = n^3/b reads
      {read block B(k,j) into fast memory}    ... b^2 × (n/b)^3 = n^3/b reads
      C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
    {write block C(i,j) back to slow memory}  ... b^2 × (n/b)^2 = n^2 writes
```



$2n^3/b + 2n^2$ reads/writes << $2n^3$ arithmetic - Faster!

Courtesy of James Demmel

Does blocked matmul attain lower bound?

- Recall: if 3 b-by-b blocks fit in fast memory of size M, then #reads/writes = $2n^3/b + 2n^2$
- Make b as large as possible: $3b^2 \leq M$, so #reads/writes $\geq 3^{1/2}n^3/M^{1/2} + 2n^2$
- Attains lower bound = Ω (#flops / $M^{1/2}$)

- But what if we don't know M?
- Or if there are multiple levels of fast memory?
- How do we write the algorithm?

Courtesy of James Demmel

Recursive Matrix Multiplication (RMM) (1/2)

- For simplicity: square matrices with $n = 2^m$
- $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

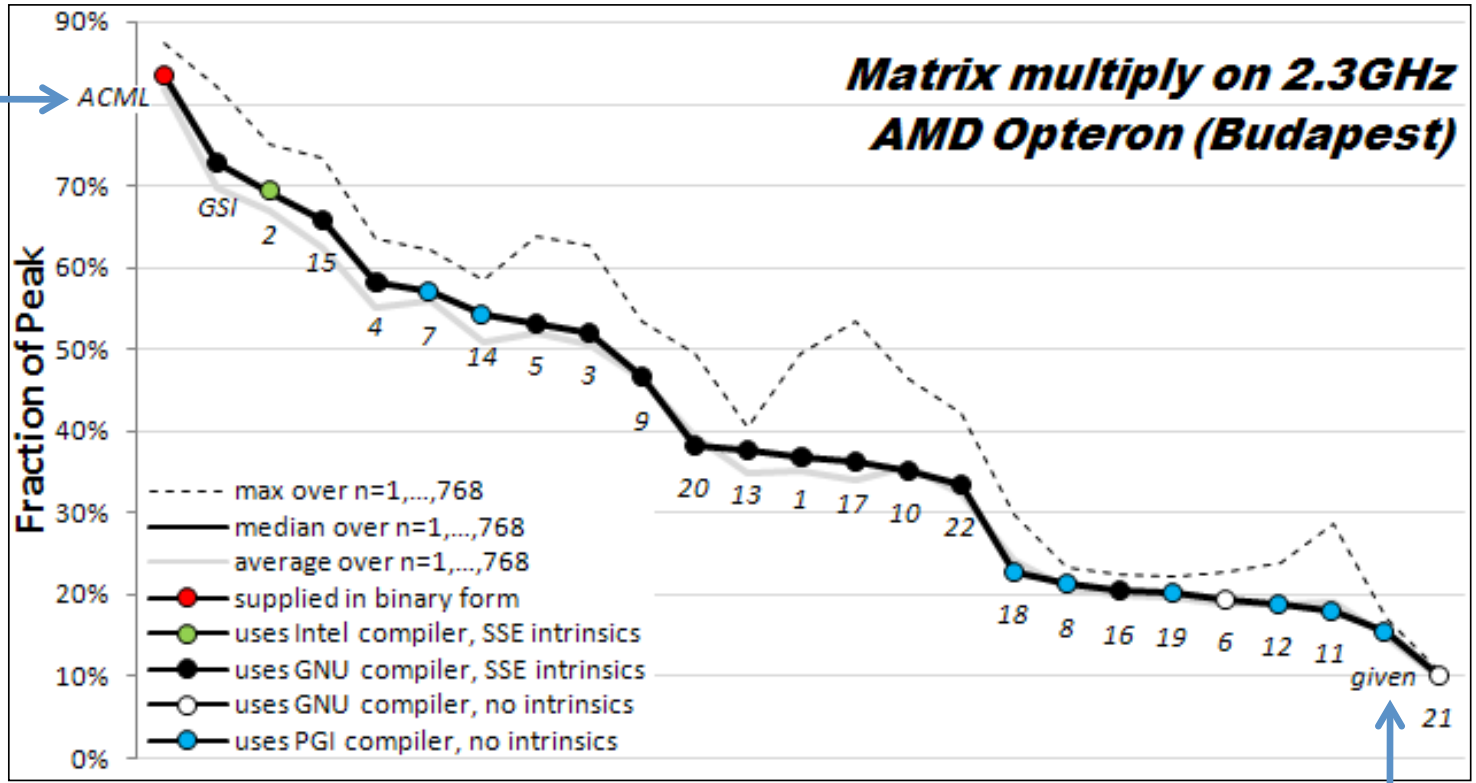
$$= \begin{pmatrix} A_{11}\cdot B_{11} + A_{12}\cdot B_{21} & A_{11}\cdot B_{12} + A_{12}\cdot B_{22} \\ A_{21}\cdot B_{11} + A_{22}\cdot B_{21} & A_{21}\cdot B_{12} + A_{22}\cdot B_{22} \end{pmatrix}$$

- True when each A_{ij} etc 1x1 or n/2 x n/2

```
func C = RMM (A, B, n)
  if n = 1, C = A * B, else
    { C11 = RMM (A11 , B11 , n/2) + RMM (A12 , B21 , n/2)
      C12 = RMM (A11 , B12 , n/2) + RMM (A12 , B22 , n/2)
      C21 = RMM (A21 , B11 , n/2) + RMM (A22 , B21 , n/2)
      C22 = RMM (A21 , B12 , n/2) + RMM (A22 , B22 , n/2) }
  return
```


Courtesy of James Demmel

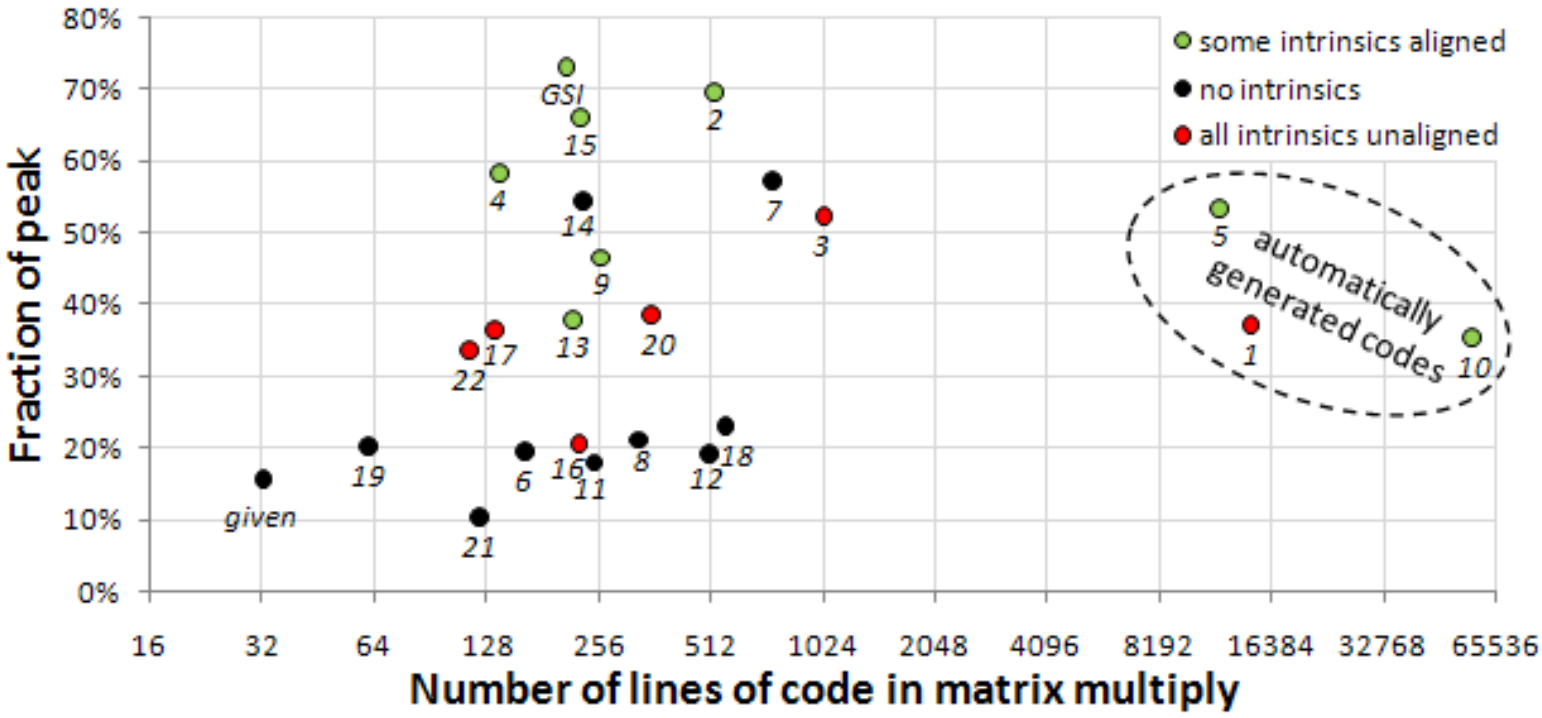
How hard is hand-tuning matmul, anyway?



- **Results of 22 student teams trying to tune matrix-multiply, in CS267 Spr09**
- **Students given "blocked" code to start with (7x faster than naïve)**
  - **Still hard to get close to vendor tuned performance (ACML) (another 6x)**
- **For more discussion, see www.cs.berkeley.edu/~volkov/cs267.sp09/hw1/results/**

Courtesy of James Demmel

How hard is hand-tuning matmul, anyway?



Courtesy of James Demmel

Solving science problems faster

Parallel computers can solve **bigger** problems

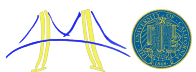
- **weak scaling**

Parallel computers can also solve a fixed problem **faster**

- **strong scaling**

Obstacles to strong scaling

- may increase relative cost of **communication**
- may hurt load balance



Courtesy of James Demmel

Achieving strong scaling

How to reduce communication and maintain load balance?

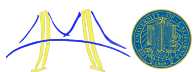
- reduce communication along the critical path

Communicate **less**

- *avoid* unnecessary communication

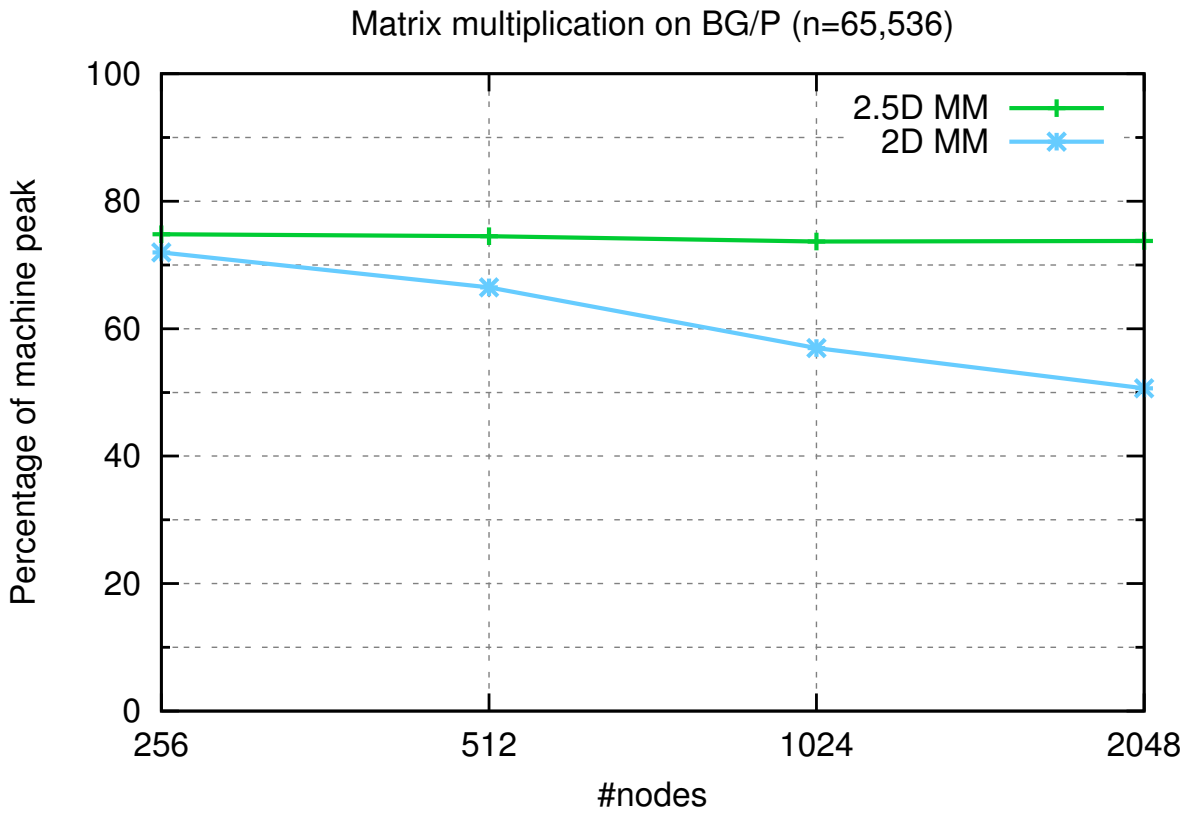
Communicate **smarter**

- know your network topology



Courtesy of James Demmel

Strong scaling matrix multiplication



Courtesy of James Demmel

Let's compute together the amount of operations and data movements

- for a 1D distribution:

Let's compute together the amount of operations and data movements

- for a 1D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	pn^2	$\frac{3n^2}{p}$

Let's compute together the amount of operations and data movements

- for a 1D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	pn^2	$\frac{3n^2}{p}$

- for a 2D distribution:

Let's compute together the amount of operations and data movements

- for a 1D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	pn^2	$\frac{3n^2}{p}$

- for a 2D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	$\sqrt{p}n^2$	$\frac{3n^2}{p}$

Let's compute together the amount of operations and data movements

- for a 1D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	pn^2	$\frac{3n^2}{p}$

- for a 2D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	$\sqrt{p}n^2$	$\frac{3n^2}{p}$

- for a 3D distribution:

  Not always that much memory available...

Let's compute together the amount of operations and data movements

- for a 1D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	pn^2	$\frac{3n^2}{p}$

- for a 2D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	$\sqrt{p}n^2$	$\frac{3n^2}{p}$

- for a 3D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	$\sqrt[3]{p}n^2$	$\frac{3n^2}{p^{2/3}}$

Not always that much memory available...

Let's compute together the amount of operations and data movements

- for a 1D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	pn^2	$\frac{3n^2}{p}$

- for a 2D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	$\sqrt{p}n^2$	$\frac{3n^2}{p}$

- for a 3D distribution:

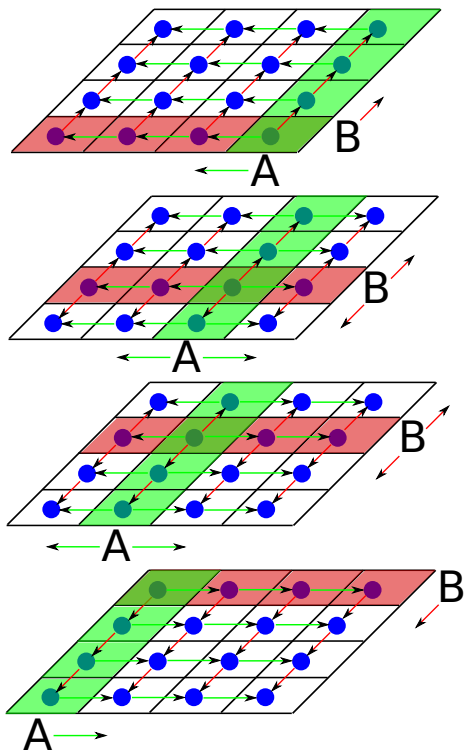
Flops	Bytes	Memory
$\frac{n^3}{p}$	$\sqrt[3]{p}n^2$	$\frac{3n^2}{p^{2/3}}$

  Not always that much memory available...

- for a 2.5D distribution:

Flops	Bytes	Memory
$\frac{n^3}{p}$	$\sqrt{\frac{p}{c}}n^2$	$\frac{3cn^2}{p}$

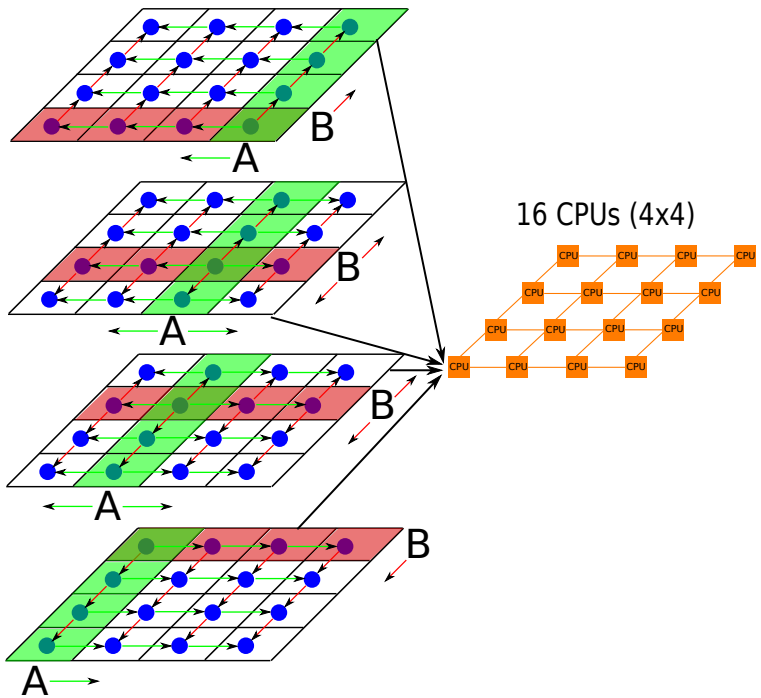
Blocking matrix multiplication

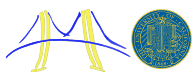


Courtesy of James Demmel

2D matrix multiplication

[Cannon 69], [Van De Geijn and Watts 97]





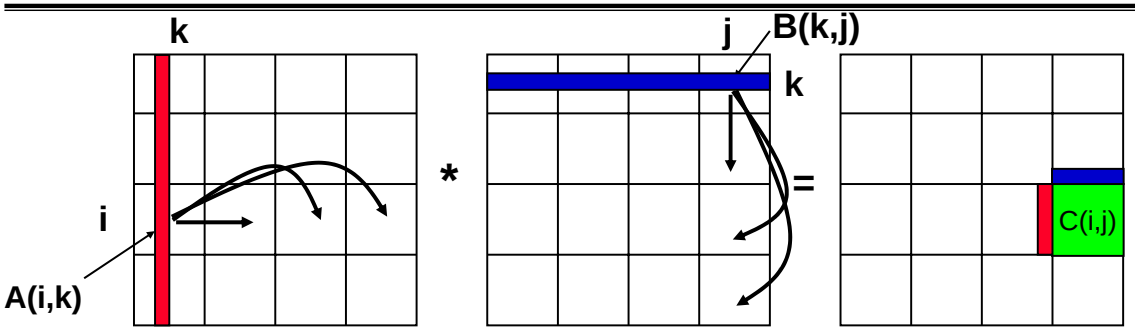
Courtesy of James Demmel

SUMMA Algorithm

- SUMMA = Scalable Universal Matrix Multiply
- Slightly less efficient, but simpler and easier to generalize
- Presentation from van de Geijn and Watts
  - www.netlib.org/lapack/lawns/lawn96.ps
  - Similar ideas appeared many times
- Used in practice in PBLAS = Parallel BLAS
  - www.netlib.org/lapack/lawns/lawn100.ps

Courtesy of James Demmel

SUMMA

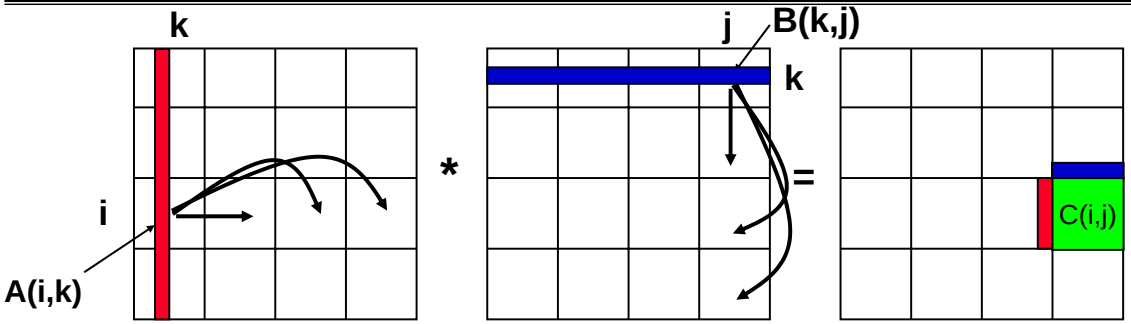


- i, j represent all rows, columns owned by a processor
- k is a single row or column
  - or a block of b rows or columns

- C(i,j) = C(i,j) + $\Sigma_{\mathbf{k}}$ A(i,k) * B(k,j)

- Assume a p_r by p_c processor grid ($p_r = p_c = 4$ above)
  - Need not be square

SUMMA



For k=0 to n-1 … or n/b-1 where b is the block size

… = # cols in A(i,k) and # rows in B(k,j)

for all i = 1 to p_r … in parallel

owner of A(i,k) broadcasts it to whole processor row

for all j = 1 to p_c … in parallel

owner of B(k,j) broadcasts it to whole processor column

Receive A(i,k) into Acol

Receive B(k,j) into Brow

C_myproc = C_myproc + Acol * Brow

Courtesy of James Demmel

SUMMA performance

- To simplify analysis only, assume s = sqrt(p)

```
For k=0 to n/b-1
    for all i = 1 to s   ...  s = sqrt(p)
        owner of A(i,k) broadcasts it to whole processor row
            ... time = log s *( α + β * b*n/s), using a tree
    for all j = 1 to  s
        owner of B(k,j) broadcasts it to whole processor column
            ... time = log s *( α + β * b*n/s), using a tree
    Receive A(i,k) into Acol
    Receive B(k,j) into Brow
    C_myproc = C_myproc + Acol * Brow
            ... time = 2*(n/s)^2*b
```

- Total time = $2*n^3/p + \alpha * \log p * n/b + \beta * \log p * n^2/s$

SUMMA performance

- Total time = $2*n^3/p + \alpha * \log p * n/b + \beta * \log p * n^2/s$
- Parallel Efficiency =

  $1/(1 + \alpha * \log p * p / (2*b*n^2) + \beta * \log p * s/(2*n))$
- ~Same β term as Cannon, except for log p factor

  log p grows slowly so this is ok
- Latency (α) term can be larger, depending on b

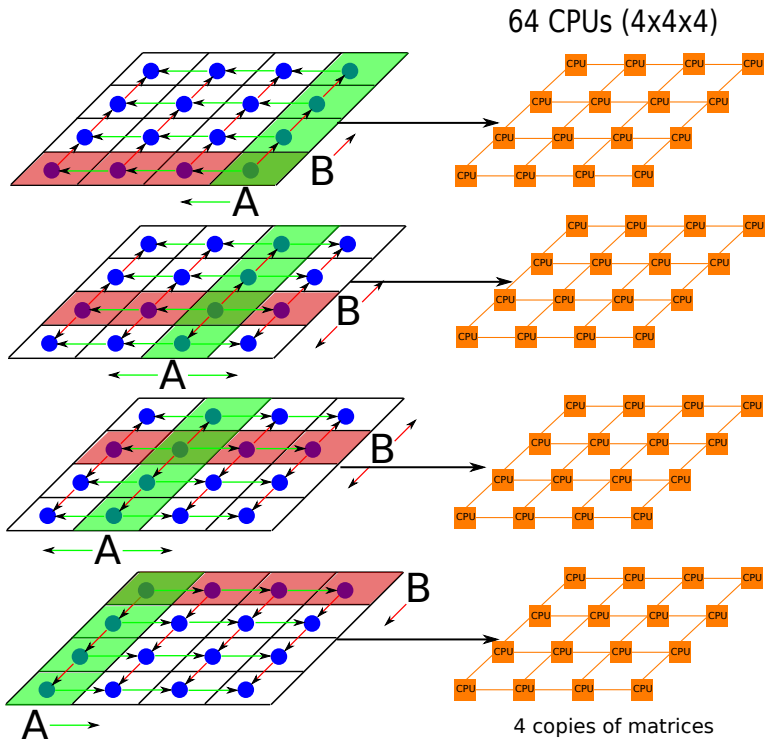
  When b=1, get $\alpha * \log p * n$

  As b grows to n/s, term shrinks to

  $\alpha * \log p * s$ (log p times Cannon)
- Temporary storage grows like 2*b*n/s
- Can change b to tradeoff latency cost with memory

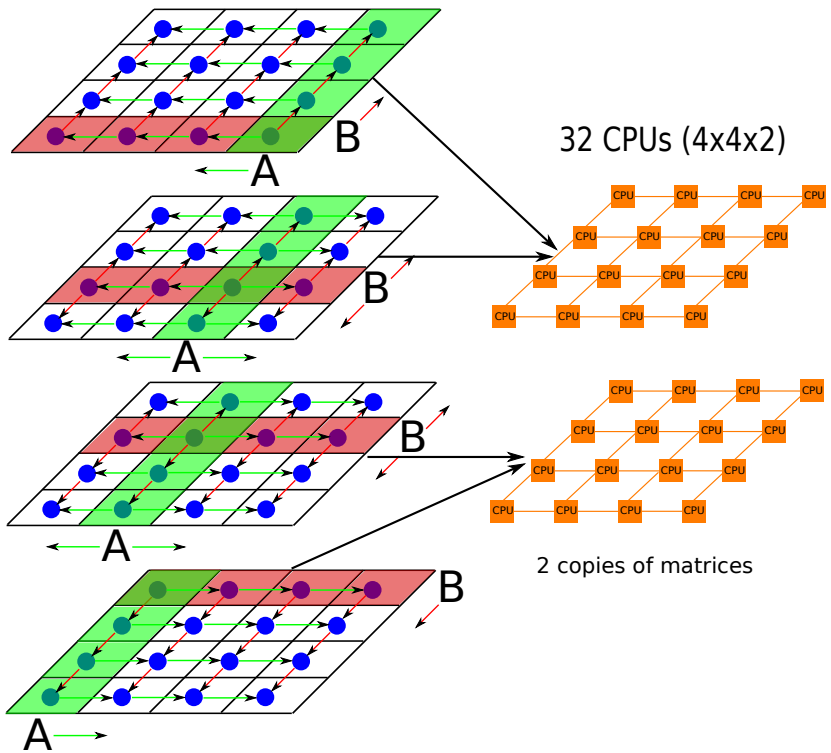
3D matrix multiplication

[Agarwal et al 95], [Aggarwal, Chandra, and Snir 90], [Bernsten 89]



Courtesy of James Demmel

2.5D matrix multiplication



Courtesy of James Demmel

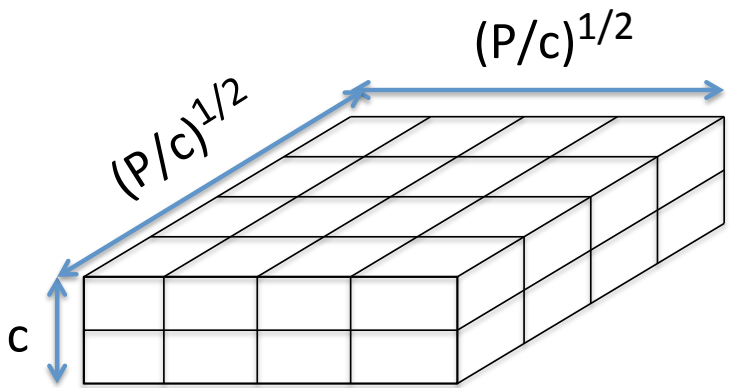
Can we do better?

- Lower bound assumed 1 copy of data: $M = O(n^2/P)$ per proc.
- What if matrix small enough to fit c>1 copies, so $M = cn^2/P$?
  - #words_moved = $\Omega(\text{ \#flops } / M^{1/2}) = \Omega(n^2 / (c^{1/2} P^{1/2}))$
  - #messages = $\Omega(\text{ \#flops } / M^{3/2}) = \Omega(P^{1/2} / c^{3/2})$
- Can we attain new lower bound?
  - Special case: "3D Matmul": $c = P^{1/3}$
    - Dekel, Nassimi, Sahni [81], Bernsten [89], Agarwal, Chandra, Snir [90], Johnson [93], Agarwal, Balle, Gustavson, Joshi, Palkar [95]
    - Processors arranged in $P^{1/3} \times P^{1/3} \times P^{1/3}$ grid
    - Processor (i,j,k) performs C(i,j) = C(i,j) + A(i,k)*B(k,j), where each submatrix is $n/P^{1/3} \times n/P^{1/3}$
  - Not always that much memory available…

Courtesy of James Demmel

2.5D Matrix Multiplication

- Assume can fit cn^2/P data per processor, $c>1$
- Processors form $(P/c)^{1/2}$ x $(P/c)^{1/2}$ x c grid

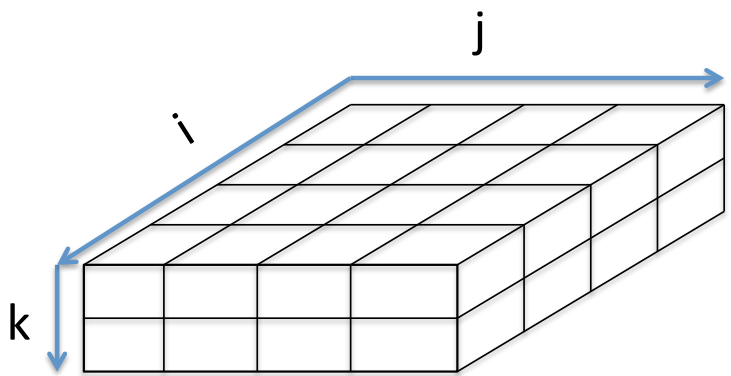


Example: P = 32, c = 2

Courtesy of James Demmel

2.5D Matrix Multiplication

- Assume can fit cn^2/P data per processor, $c > 1$
- Processors form $(P/c)^{1/2}$ x $(P/c)^{1/2}$ x c grid



Initially P(i,j,0) owns A(i,j) and B(i,j)
each of size $n(c/P)^{1/2}$ x $n(c/P)^{1/2}$

(1) P(i,j,0) broadcasts A(i,j) and B(i,j) to P(i,j,k)
(2) Processors at level k perform 1/c-th of SUMMA, i.e. 1/c-th of Σ_m A(i,m)*B(m,j)
(3) Sum-reduce partial sums Σ_m A(i,m)*B(m,j) along k-axis so P(i,j,0) owns C(i,j)

Courtesy of James Demmel

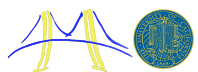
2.5D strong scaling

n = dimension, p = #processors, c = #copies of data

- must satisfy $1 \leq c \leq p^{1/3}$
- special case: $c = 1$ yields 2D algorithm
- special case: $c = p^{1/3}$ yields 3D algorithm

$$\text{cost(2.5D MM}(p, c)) = O(n^3/p) \text{ flops} + O(n^2/\sqrt{c \cdot p}) \text{ words moved} + O(\sqrt{p/c^3}) \text{ messages}^*$$

*ignoring log(p) factors



Courtesy of James Demmel

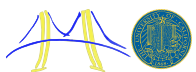
2.5D strong scaling

n = dimension, p = #processors, c = #copies of data

- must satisfy $1 \leq c \leq p^{1/3}$
- special case: $c = 1$ yields 2D algorithm
- special case: $c = p^{1/3}$ yields 3D algorithm

$$\begin{aligned} \text{cost(2D MM}(p)) &= O(n^3/p) \text{ flops} \\ &+ O(n^2/\sqrt{p}) \text{ words moved} \\ &+ O(\sqrt{p}) \text{ messages}^* \\ &= \text{cost(2.5D MM}(p, 1)) \end{aligned}$$

*ignoring log(p) factors



Courtesy of James Demmel

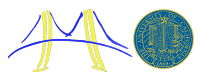
2.5D strong scaling

n = dimension, p = #processors, c = #copies of data

- must satisfy $1 \leq c \leq p^{1/3}$
- special case: $c = 1$ yields 2D algorithm
- special case: $c = p^{1/3}$ yields 3D algorithm

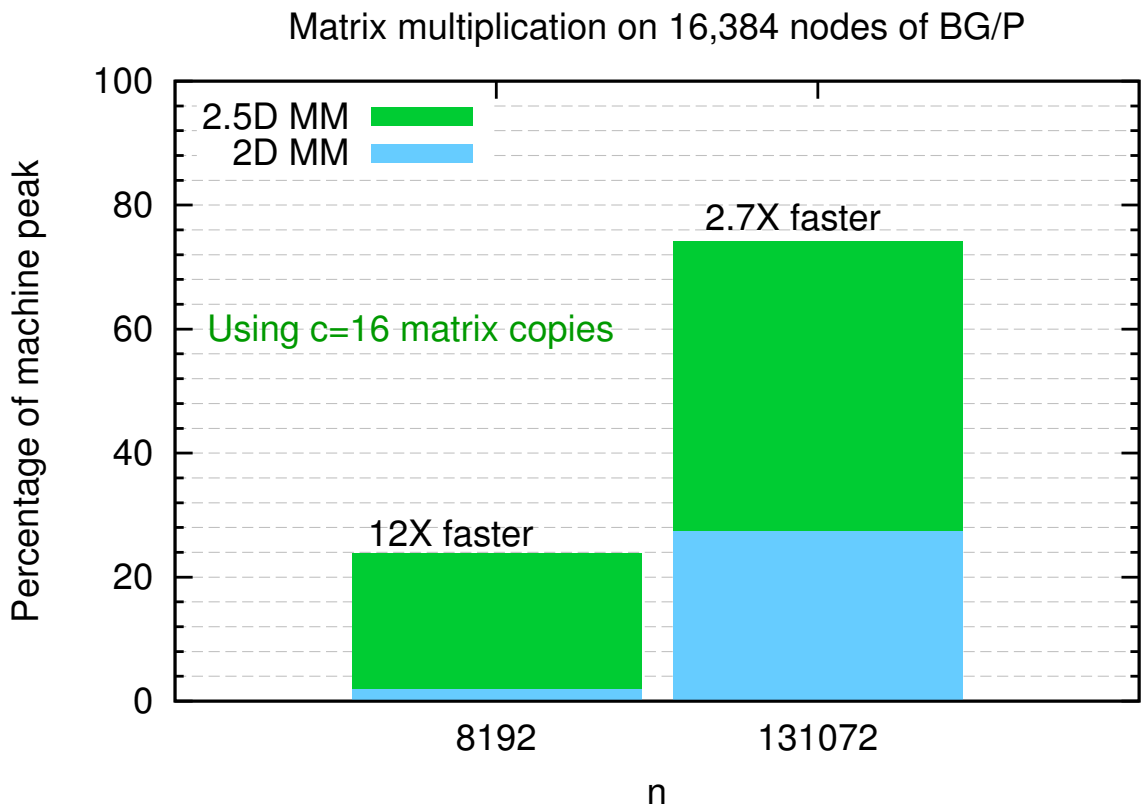
$$\begin{aligned}\text{cost}(\text{2.5D MM}(\mathbf{c} \cdot p, \mathbf{c})) = & \; O(n^3/(\mathbf{c} \cdot p)) \text{ flops} \\ & + O(n^2/(\mathbf{c} \cdot \sqrt{p})) \text{ words moved} \\ & + O(\sqrt{p}/\mathbf{c}) \text{ messages} \\ = & \; \text{cost}(\text{2D MM}(p))/\mathbf{c}\end{aligned}$$

perfect strong scaling



Courtesy of James Demmel

2.5D MM on 65,536 cores

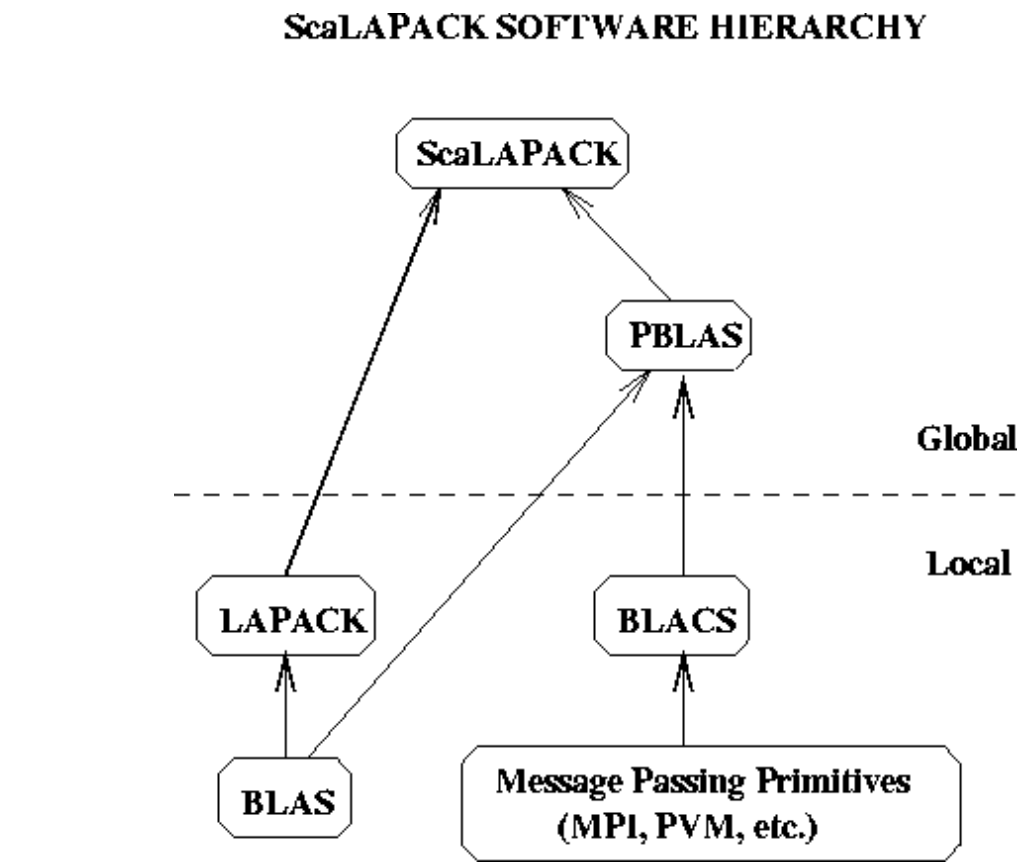


Courtesy of James Demmel

Outline

1. Communication "Avoiding" Algorithms
   - Cache oblivious algorithm
   - Parallel Algorithm
2. Synchronization-reducing algorithms
   - DAG generation
   - DAG generation
   - Granularity and Hybrid Computing
3. Auto-tuning
4. Reproducibility and Mixed-Precision Methods

ScaLAPACK Parallel Library



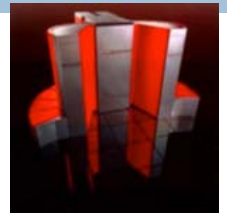
Courtesy of James Demmel

LAPACK and ScaLAPACK

	LAPACK	ScaLAPACK
Machines	Workstations, Vector, SMP	Distributed Memory, DSM
Based on	BLAS	BLAS, BLACS
Functionality	Linear Systems Least Squares Eigenproblems	Linear Systems Least Squares Eigenproblems (less than LAPACK)
Matrix types	Dense, band	Dense, band, out-of-core
Error Bounds	Complete	A few
Languages	F77 or C	F77 and C
Interfaces to	C++, F90	HPF
Manual?	Yes	Yes
Where?	www.netlib.org/lapack	www.netlib.org/scalapack

Courtesy of James Demmel

Looking at the Gordon Bell Prize



- 1 GFlop/s; 1988; Cray Y-MP; 8 Processors
  - Static finite element analysis

- 1 TFlop/s; 1998; Cray T3E; 1024 Processors
  - Modeling of metallic magnet atoms, using a variation of the locally self-consistent multiple scattering method.

- 1 PFlop/s; 2008; Cray XT5; 1.5×10^5 Processors
  - Superconductive materials
- 1 EFlop/s; ~2018; ?; 1×10^7 Processors (10^9 threads)

Courtesy of Jack Dongarra

Major Changes to Software

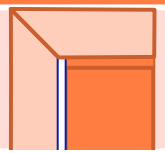
- **Must rethink the design of our software**
  - **Another disruptive technology**
    - **Similar to what happened with cluster computing and message passing**
  - **Rethink and rewrite the applications, algorithms, and software**
- **Numerical libraries for example will change**
  - **For example, both LAPACK and ScaLAPACK will undergo major changes to accommodate this**

14

Courtesy of Jack Dongarra

A New Generation of Software:

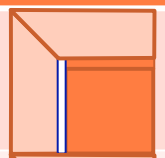
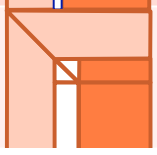
Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations

Courtesy of Jack Dongarra

A New Generation of Software:

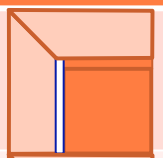
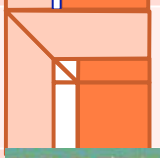
Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations

Courtesy of Jack Dongarra

A New Generation of Software:

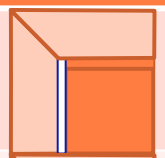
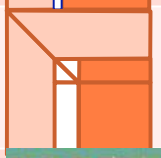
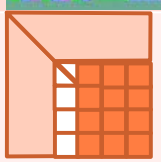
Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing

Courtesy of Jack Dongarra

A New Generation of Software:

Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing
PLASMA (00's) New Algorithms (many-core friendly)		Rely on - a DAG/scheduler - block data layout - some extra kernels

Those new algorithms
- have a very low granularity, they scale very well (multicore, petascale computing, …)
- removes a lots of dependencies among the tasks, (multicore, distributed computing)
- avoid latency (distributed computing, out-of-core)
- rely on fast kernels

Those new algorithms need new kernels and rely on efficient scheduling algorithms.

Courtesy of Jack Dongarra

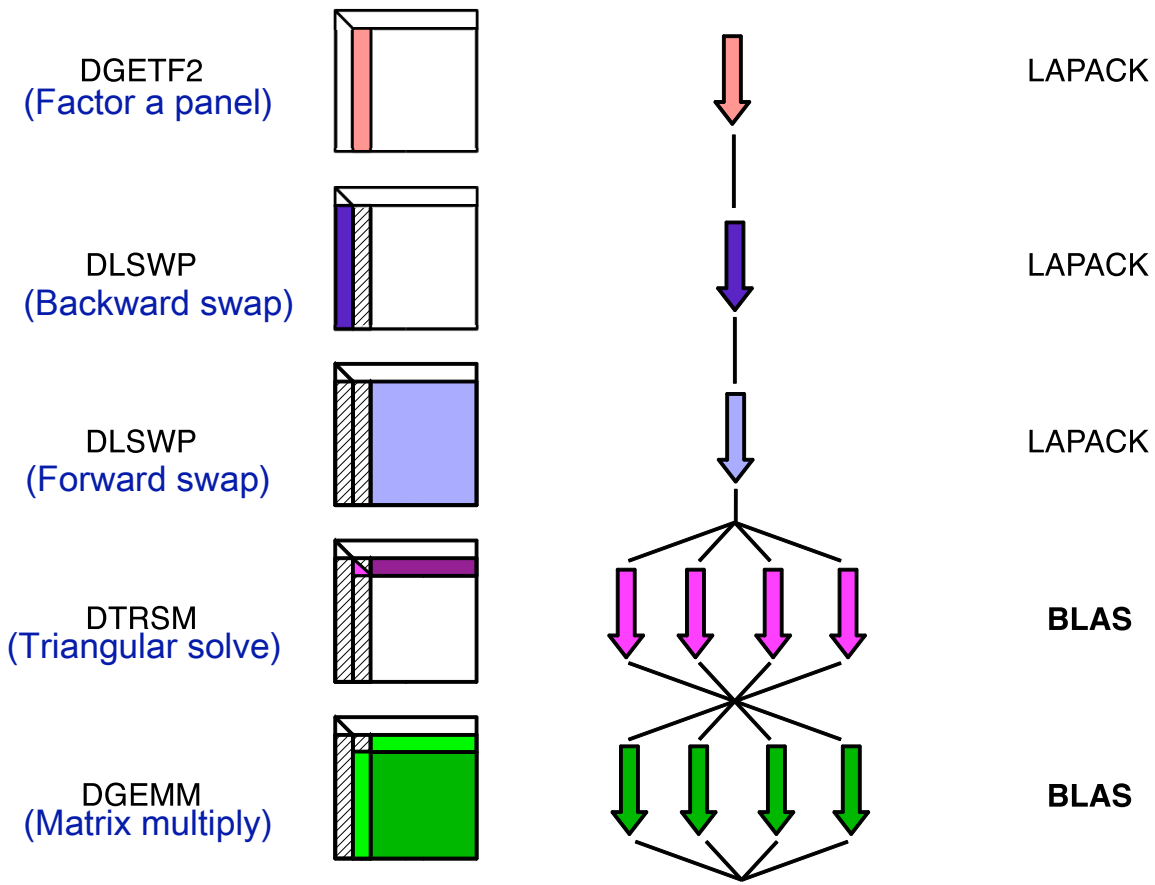
Coding for an Abstract Multicore

Parallel software for multicores should have two characteristics:

- Fine granularity:
  - High level of parallelism is needed
  - Cores will probably be associated with relatively small local memories. This requires splitting an operation into tasks that operate on small portions of data in order to reduce bus traffic and improve data locality.
- Asynchronicity:
  - As the degree of thread level parallelism grows and granularity of the operations becomes smaller, the presence of synchronization points in a parallel execution seriously affects the efficiency of an algorithm.

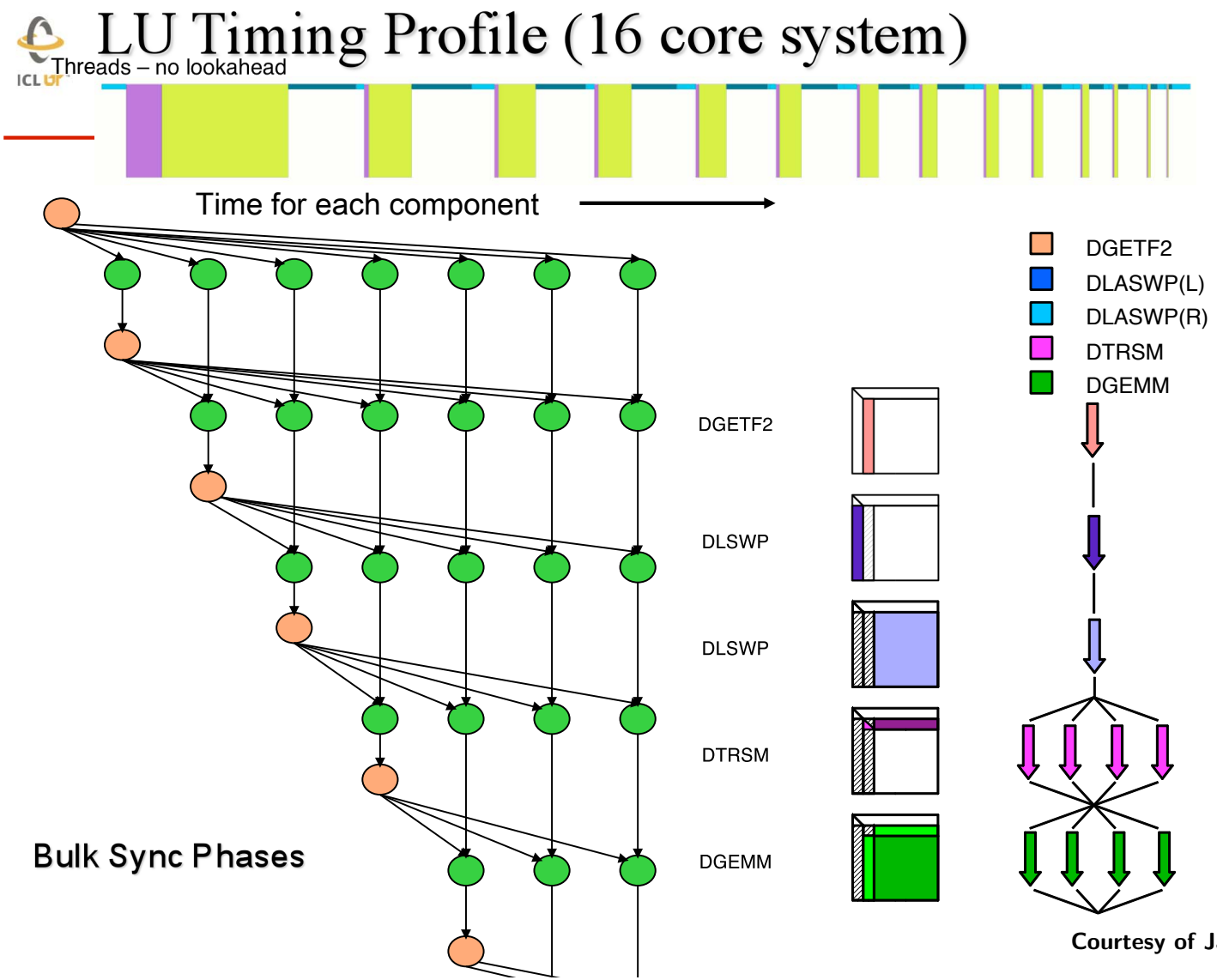
Courtesy of Jack Dongarra

Steps in the LAPACK LU



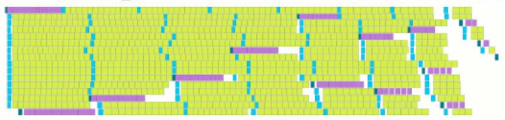
20

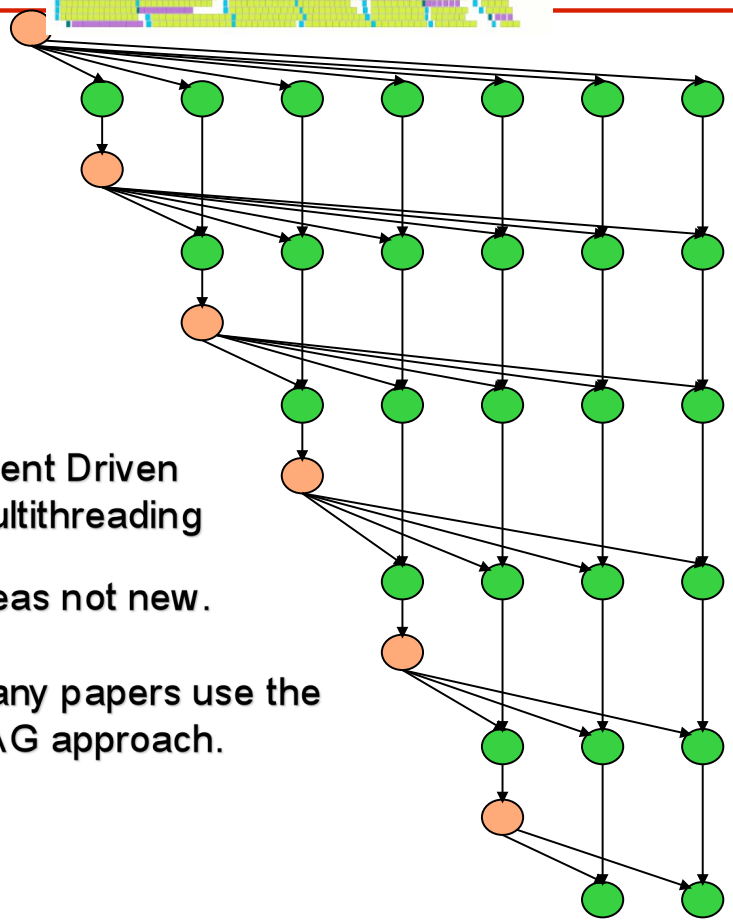
Courtesy of Jack Dongarra



Courtesy of Jack Dongarra

Adaptive Lookahead - Dynamic





Event Driven Multithreading

Ideas not new.

Many papers use the DAG approach.

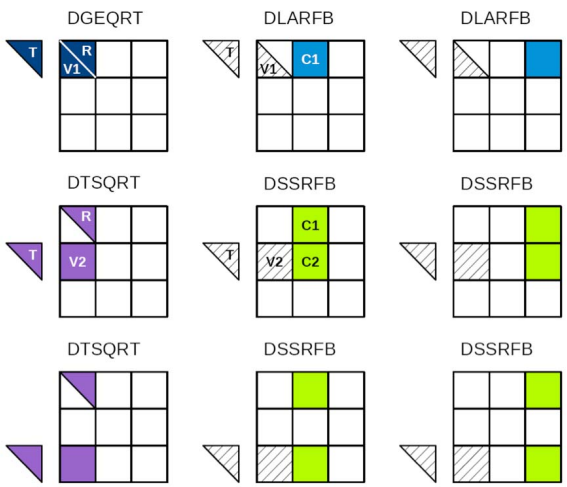
```
while(1)
   fetch_task();
   switch(task.type) {
      case PANEL:                    //
         dgetf2();
         update_progress();
      case COLUMN:                   //
         dlaswp();                   //
         dtrsm();
         dgemm();
         update_progress();
      case END:                      //
         for()
            dlaswp();
         return;
   }
}
```

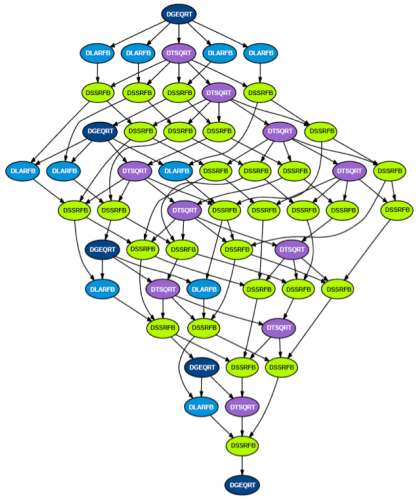
Reorganizing algorithms to use this approach

22

Courtesy of Jack Dongarra

Tile QR (&LU) Algorithms





```
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])
```

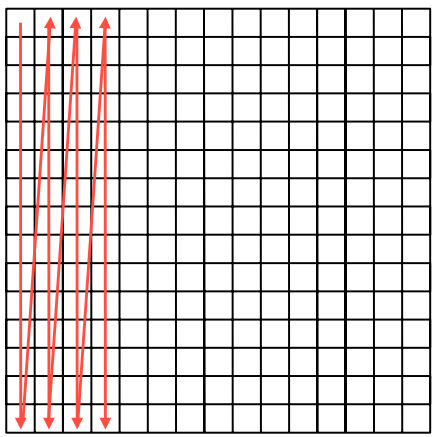
- input matrix stored and processed by square tiles
- complex DAG

Courtesy of Jack Dongarra

Achieving Fine Granularity

Fine granularity may require novel data formats to overcome the limitations of BLAS on small chunks of data.

Column-Major

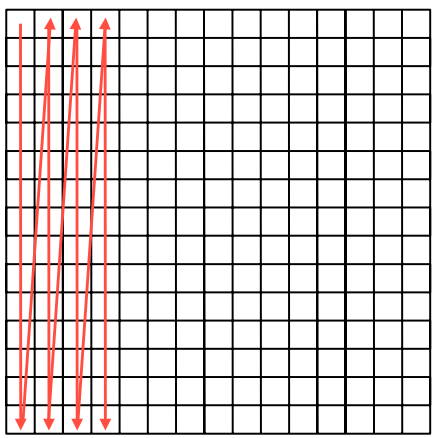


Courtesy of Jack Dongarra

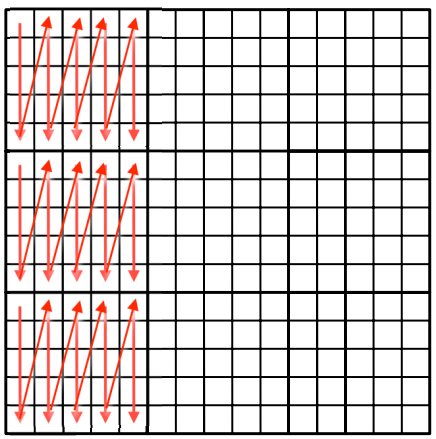
Achieving Fine Granularity

Fine granularity may require novel data formats to overcome the limitations of BLAS on small chunks of data.

Column-Major



Blocked



Courtesy of Jack Dongarra

PLASMA (Redesign LAPACK/ScaLAPACK)

Parallel Linear Algebra Software for Multicore Architectures

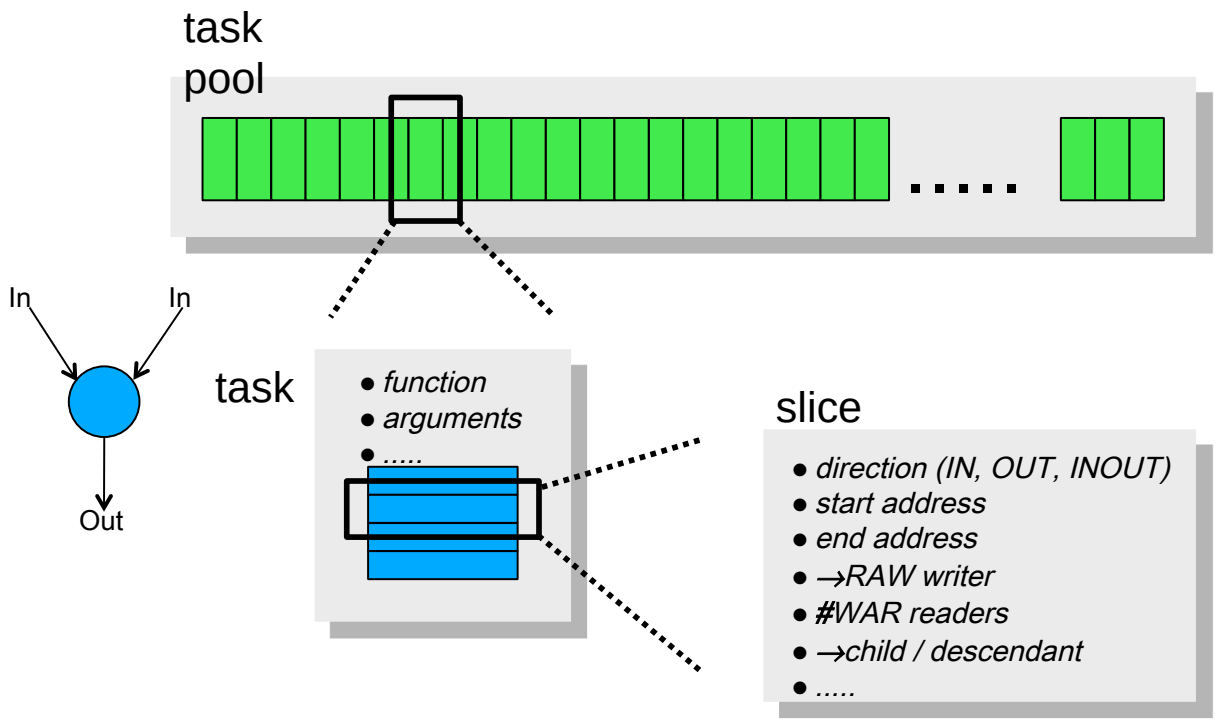
- **Asychronicity**
  - **Avoid fork-join (Bulk sync design)**
- **Dynamic Scheduling**
  - **Out of order execution**
- **Fine Granularity**
  - **Independent block operations**
- **Locality of Reference**
  - **Data storage – Block Data Layout**

Lead by Tennessee and Berkeley similar to LAPACK/ScaLAPACK as a community effort

26

Courtesy of Jack Dongarra

PLASMA Dynamic Task Scheduler

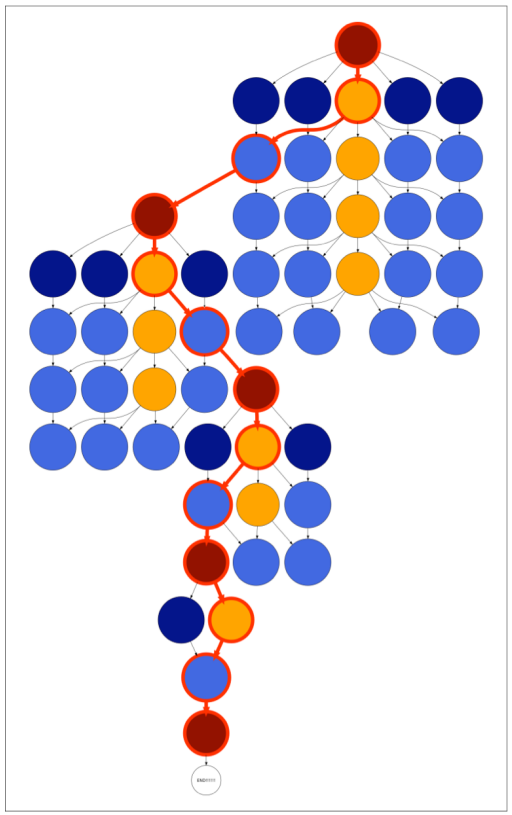


- task – a unit of scheduling (quantum of work)
- slice – a unit of dependency resolution (quantum of data)
- Current version uses one core to manage the task pool

Courtesy of Jack Dongarra

If We Had A Small Matrix Problem

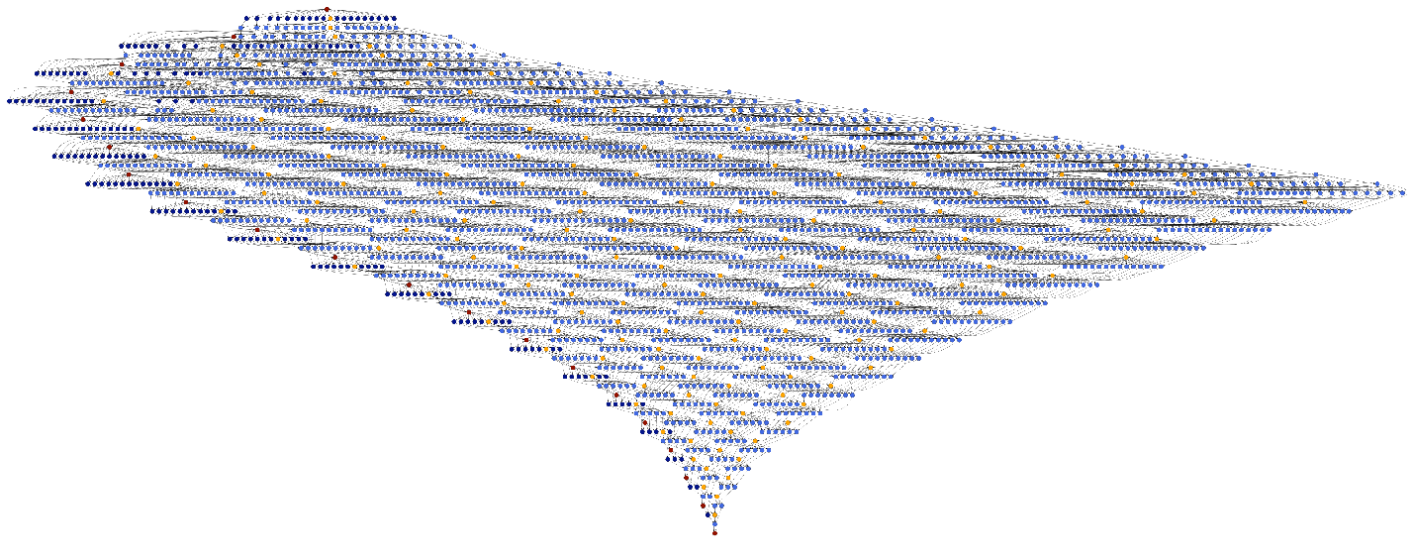
- We would generate the DAG, find the critical path and execute it.
- DAG too large to generate ahead of time
  - Not explicitly generate
  - Dynamically generate the DAG as we go
- Machines will have large number of cores in a distributed fashion
  - Will have to engage in message passing
  - Distributed management
  - Locally have a run time system



Courtesy of Jack Dongarra

The DAGs are Large

- **Here is the DAG for a factorization on a 20 x 20 matrix**

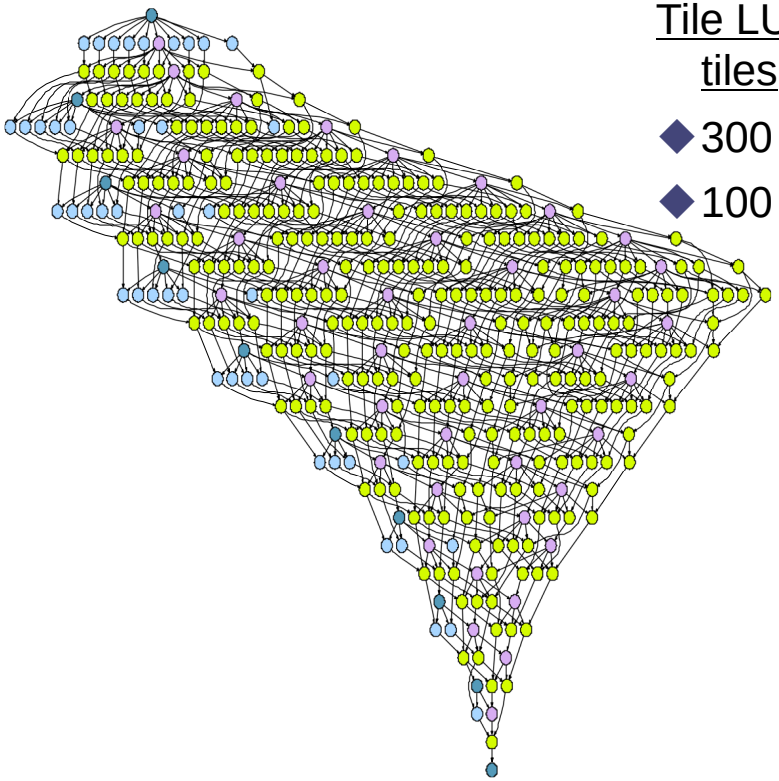


- **For a large matrix say O(10^6) the DAG is huge**
- **Many challenges for the software**

28

Courtesy of Jack Dongarra

Execution of the DAG by a Sliding Window

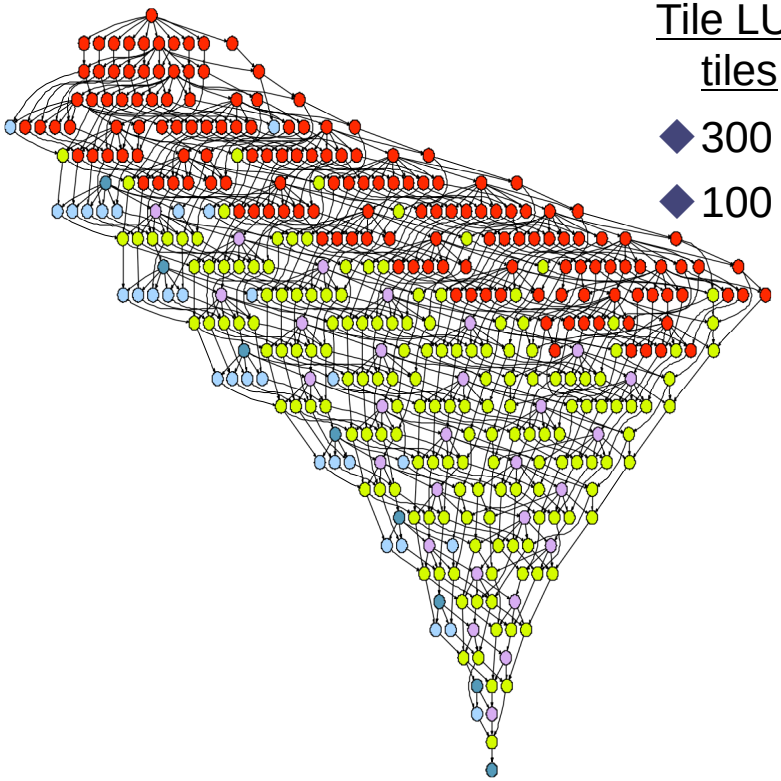


Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window

Courtesy of Jack Dongarra

Execution of the DAG by a Sliding Window

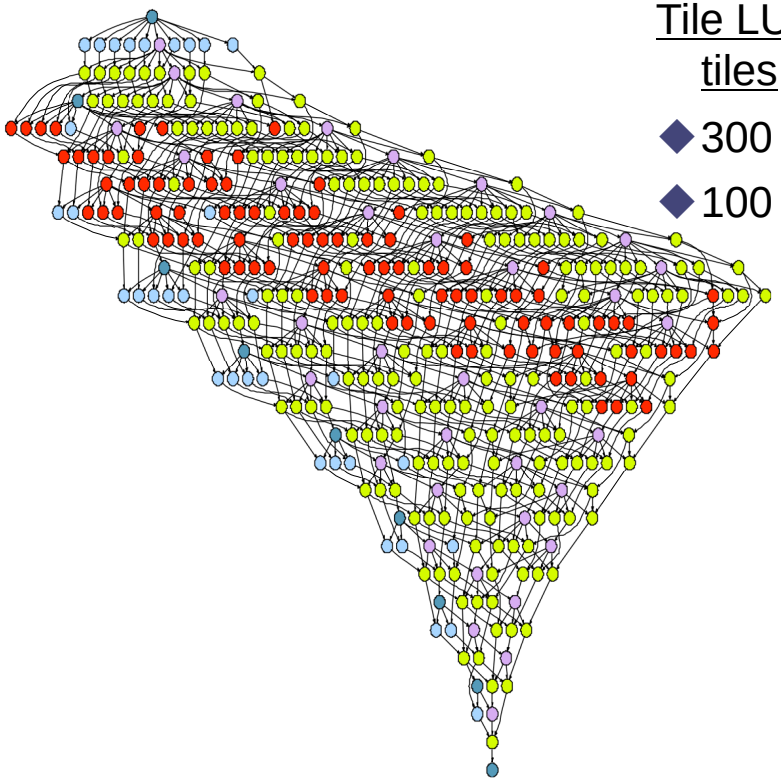


Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window

Courtesy of Jack Dongarra

Execution of the DAG by a Sliding Window

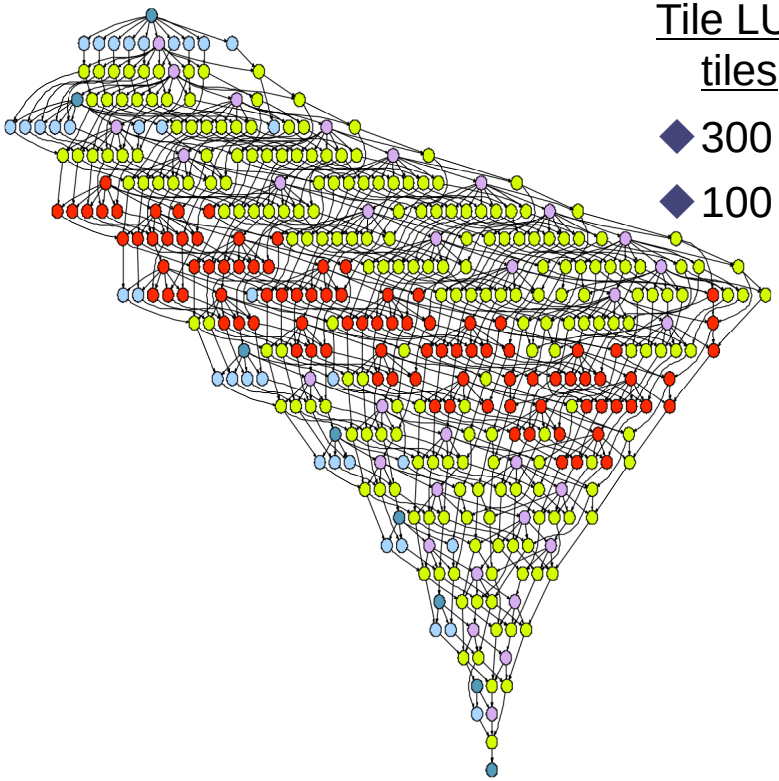


Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window

Courtesy of Jack Dongarra

Execution of the DAG by a Sliding Window

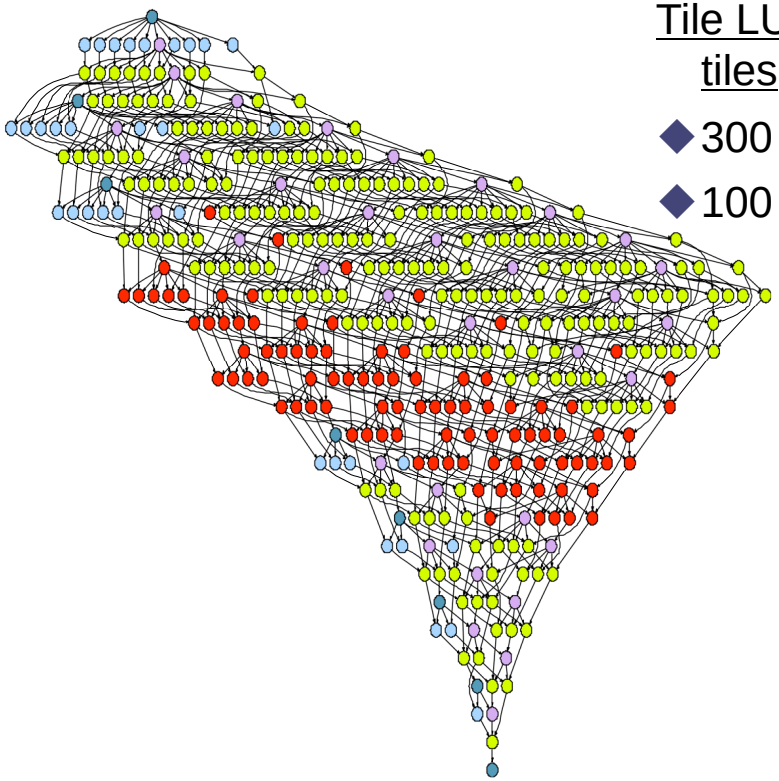


Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window

Courtesy of Jack Dongarra

Execution of the DAG by a Sliding Window

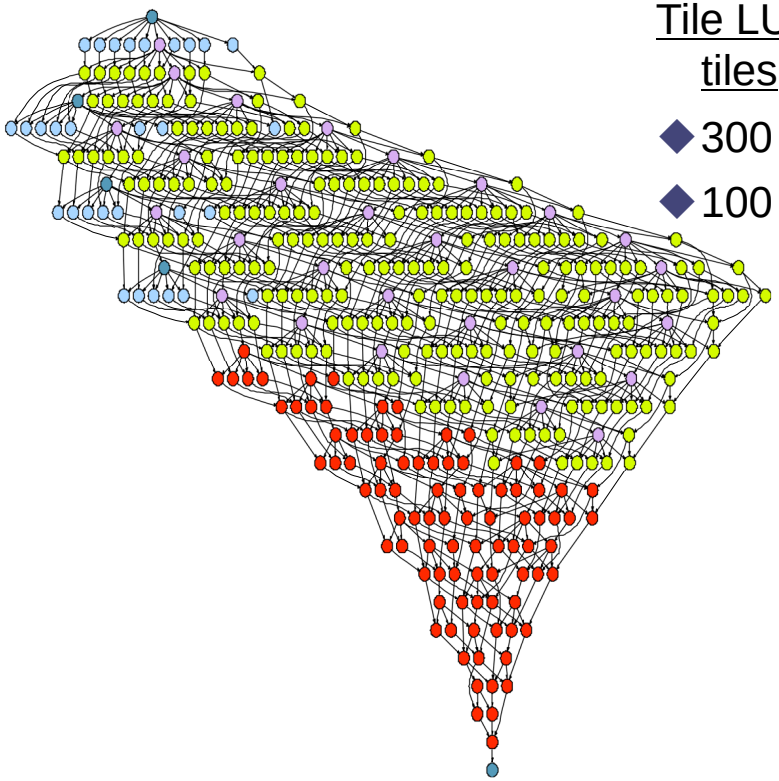


Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window

Courtesy of Jack Dongarra

Execution of the DAG by a Sliding Window



Tile LU factorization 10x10 tiles

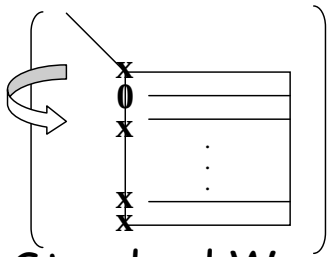
- 300 tasks total
- 100 task window

Courtesy of Jack Dongarra

Need for rewriting all algorithms as DAGs ? How to do online (and distributed?) DAG generation?

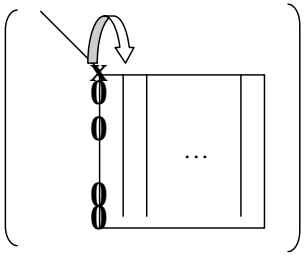
1. Bound the number of tasks and execute the sequential tasks with fake kernel calls to obtain the dependencies.
   Doing so you trade memory for scheduling opportunities. Although this approach ensures that this will be compatible with sequential execution on a semantic point of view, it also biases the execution and forces it to be close to the sequential execution [Quark/StarPU/MORSE]
2. Put the compiler in. The compiler creates the DAG at compilation time but in a compact symbolic way (i.e. a cyclic dependency graph).
   This allows to track for any task what are the child and ancestors. This helps for fault tolerance because this ensures one can reproduce any data and track down what needs to be recomputed.
3. Non-affine loops (e.g., a reduction) that do not fit in the polyhedral model are written by hands.

Gaussian Elimination



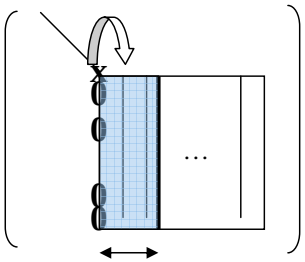
Standard Way

subtract a multiple of a row



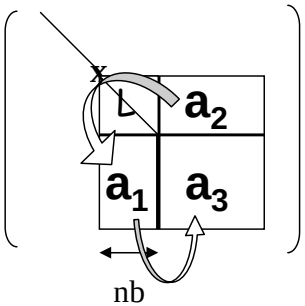
LINPACK

apply sequence to a column



LAPACK

apply sequence to nb



then apply nb to rest of matrix

$a_2 = L^{-1} a_2$

$a_3 = a_3 - a_1 * a_2$

Gaussian Elimination via a Recursive Algorithm

F. Gustavson and S. Toledo

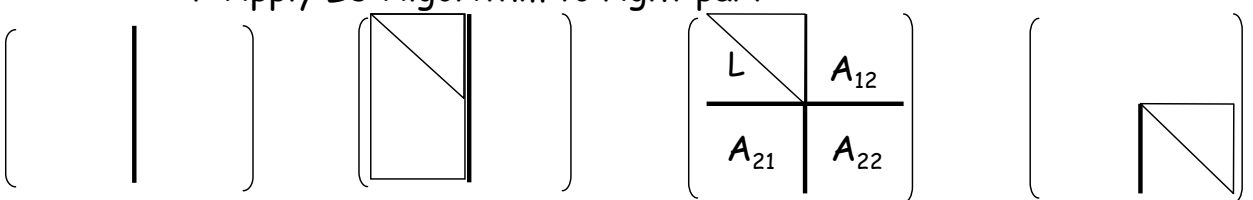
LU Algorithm:

1: Split matrix into two rectangles (m x n/2)
if only 1 column, scale by reciprocal of pivot & return

2: Apply LU Algorithm to the left part

3: Apply transformations to right part
(triangular solve $A_{12} = L^{-1}A_{12}$ and
matrix multiplication $A_{22}=A_{22} - A_{21}*A_{12}$)

4: Apply LU Algorithm to right part



Most of the work in the matrix multiply
Matrices of size n/2, n/4, n/8, …

Slide source: Dongarra

Courtesy of James Demmel

An ideal solution?

- Such dynamic/WS techniques always have trouble with data-management. Although it is possible to estimate communication costs and optimized computation kernels are stable, we end up with a greedy strategy.
- Regarding data movement optimization, sometimes, we know statically that some subDAGs could be done in an efficient way.
- They're looking at how to deal with such things. Obviously when it is recursive, adaptive computing is much easier but from classical sequential description it's more tricky.

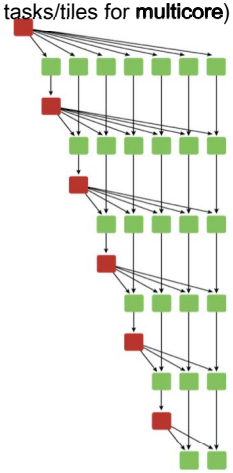
How to pick tile size ?

- When tiles are too small, bad efficiency but when too large, you do not have enough tiles, hence not enough parallelism.
- Tile size depends on hardware but when having GPUs and CPUs, this means that this choice should be done at runtime, making opportunistic scheduling choices (MAGMA, StarPU, ...).

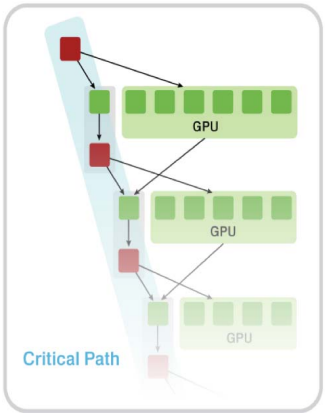
Hybrid Computing

- **Match algorithmic requirements to architectural strengths of the hybrid components**
  Multicore : small tasks/tiles
  Accelerator: large data parallel tasks

Algorithms as DAGs
(small tasks/tiles for **multicore**)



Current hybrid CPU+GPU algorithms
(small tasks for multicores and large tasks for GPUs)

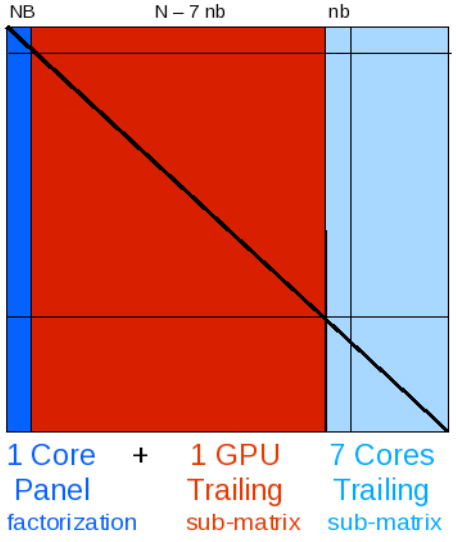


- **e.g. split the computation into tasks; define critical path that "clears" the way for other large data parallel tasks; proper schedule the tasks execution**
- **Design algorithms with well defined "*search space*" to facilitate auto-tuning**

Courtesy of Jack Dongarra

Current Work: MAGMA

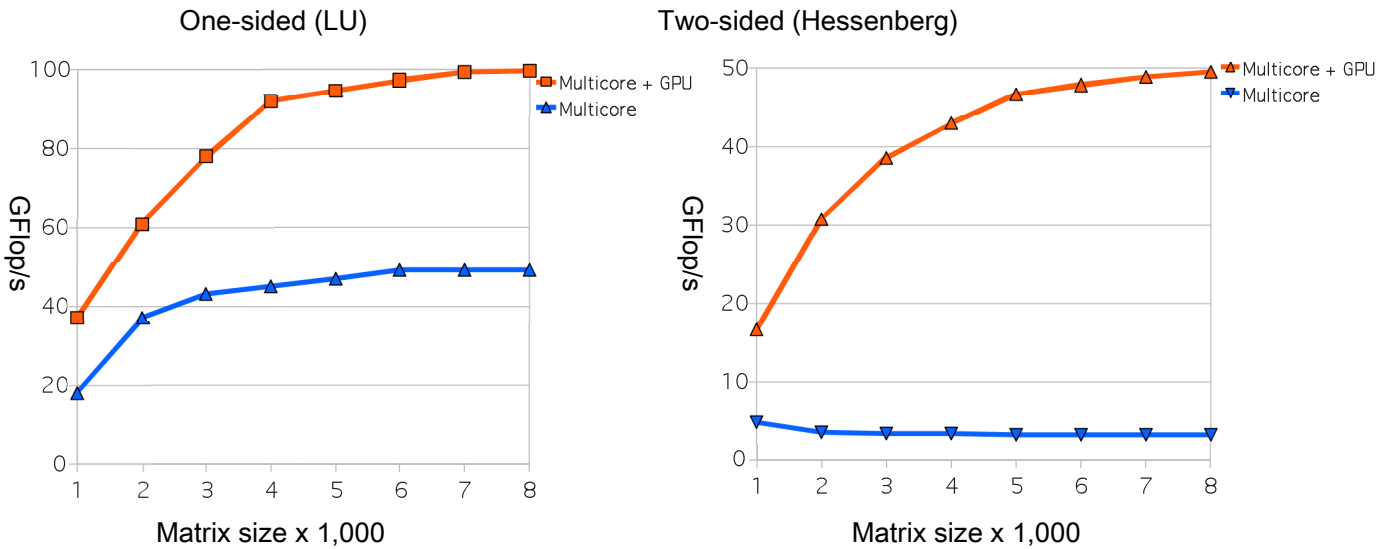
- **Algorithms (in particular LU) for Multicore + GPU systems**
- **Challenges**
  - **How to split the computation**
  - **Software development**
  - **Tuning**



Work splitting
(for single GPU + 8 cores host)

Courtesy of Jack Dongarra

Performance [in double precision]



- **Needed tuned parameters and tuned DGEMM for "rectangular" matrices**

GPU : GeForce GTX 280
(240 Cores @ 1.30 GHz)
Multicore : Intel Xeon
(2x4 Cores @ 2.33 GHz)

Courtesy of Jack Dongarra

Outline

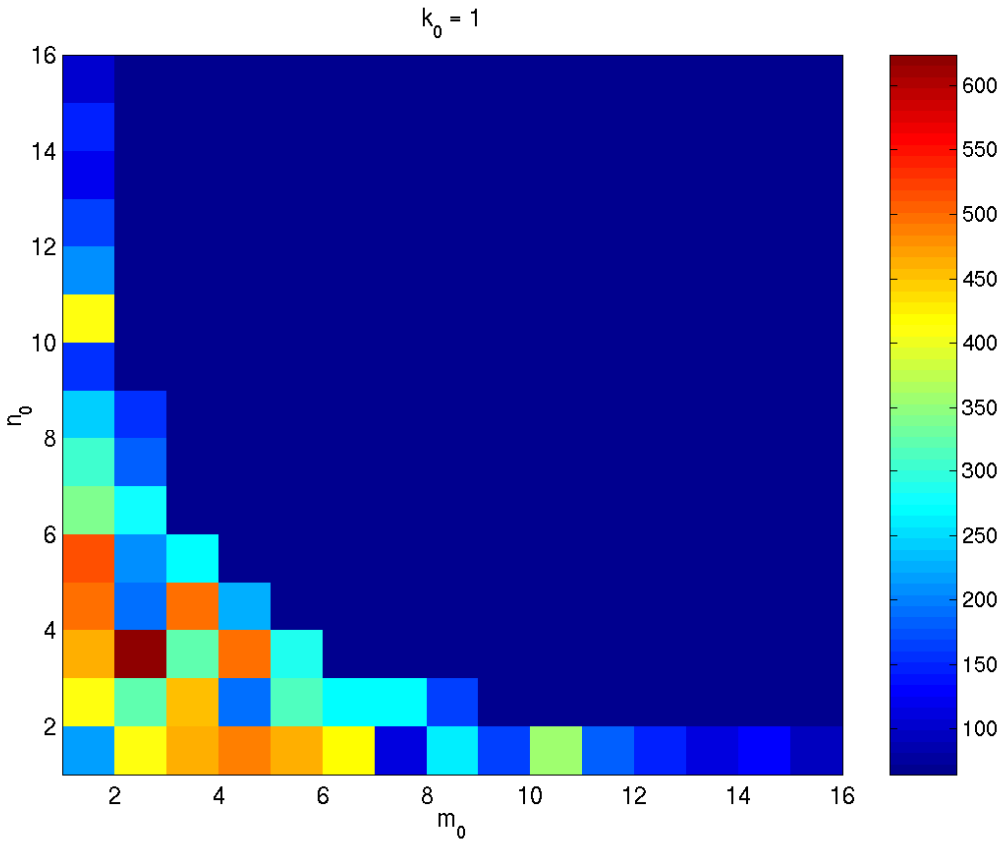
1. Communication "Avoiding" Algorithms
   - Cache oblivious algorithm
   - Parallel Algorithm
2. Synchronization-reducing algorithms
   - DAG generation
   - DAG generation
   - Granularity and Hybrid Computing
3. Auto-tuning
4. Reproducibility and Mixed-Precision Methods

Goal 3 – Automate Performance Tuning

- Widely used in performance tuning of Kernels
  - ATLAS (PhiPAC) – BLAS - www.netlib.org/atlas
  - FFTW – Fast Fourier Transform – www.fftw.org
  - Spiral – signal processing - www.spiral.net
  - OSKI – Sparse BLAS – bebop.cs.berkeley.edu/oski

Courtesy of Jack Dongarra

Optimizing blocksizes for mat-mul



Finding a Needle in a Haystack – So Automate

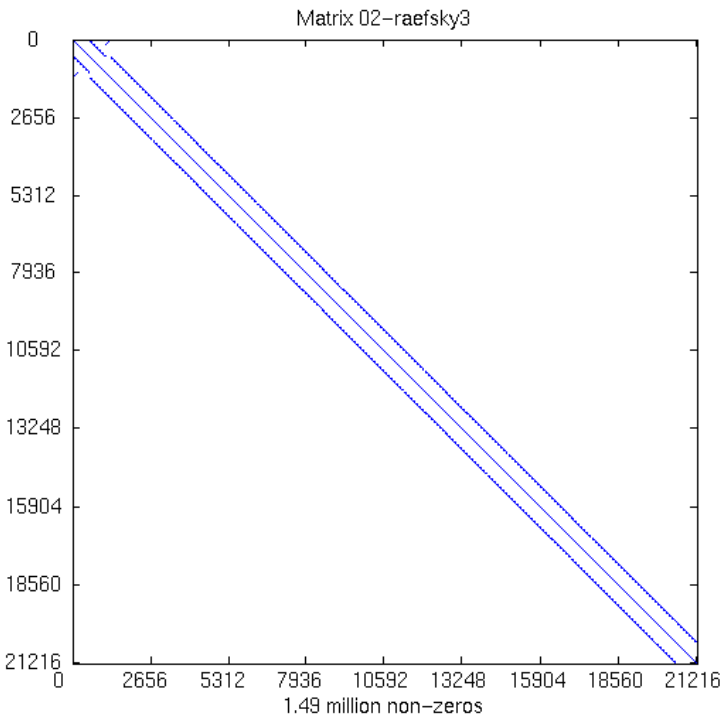
Courtesy of Jack Dongarra

Goal 3 – Automate Performance Tuning

- Widely used in performance tuning of Kernels
- 1300 calls to ILAENV() to get block sizes, etc.
  - Never been systematically tuned
- Extend automatic tuning techniques of ATLAS, etc. to these other parameters
  - Automation important as architectures evolve
- Convert ScaLAPACK data layouts on the fly
  - Important for ease-of-use too

Courtesy of Jack Dongarra

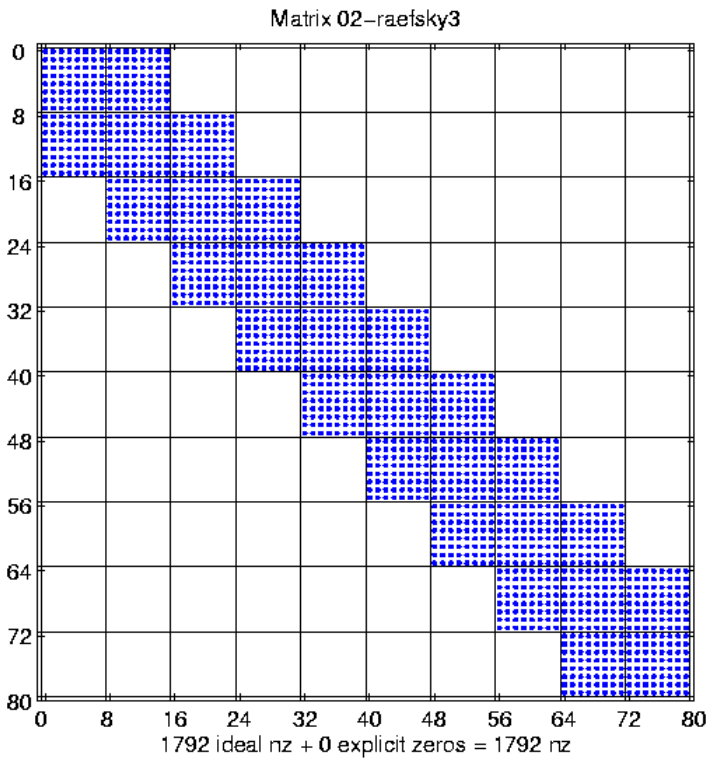
The Difficulty of Tuning SpMV: Sparse Matrix Vector Multiply




```
// y <-- y + A*x
for all A(i,j):
  y(i) += A(i,j) * x(j)
```

Courtesy of Jack Dongarra

The Difficulty of Tuning SpMV

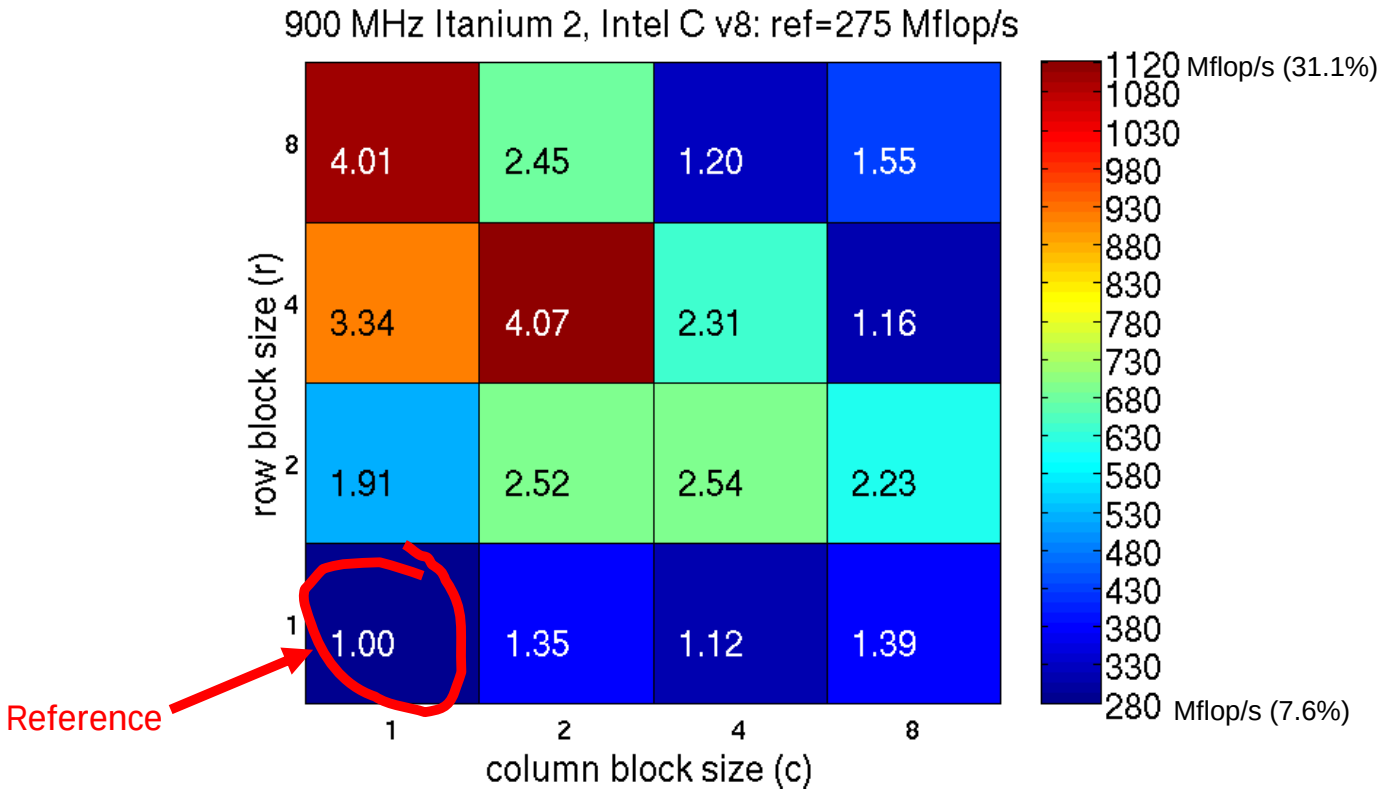



```
// y <-- y + A*x
for all A(i,j):
  y(i) += A(i,j) * x(j)

// Compressed sparse row (CSR)
for each row i:
  t = 0
  for k=row[i] to row[i+1]-1:
    t += A[k] * x[J[k]]
  y[i] = t
```

- **Exploit 8x8 dense blocks**

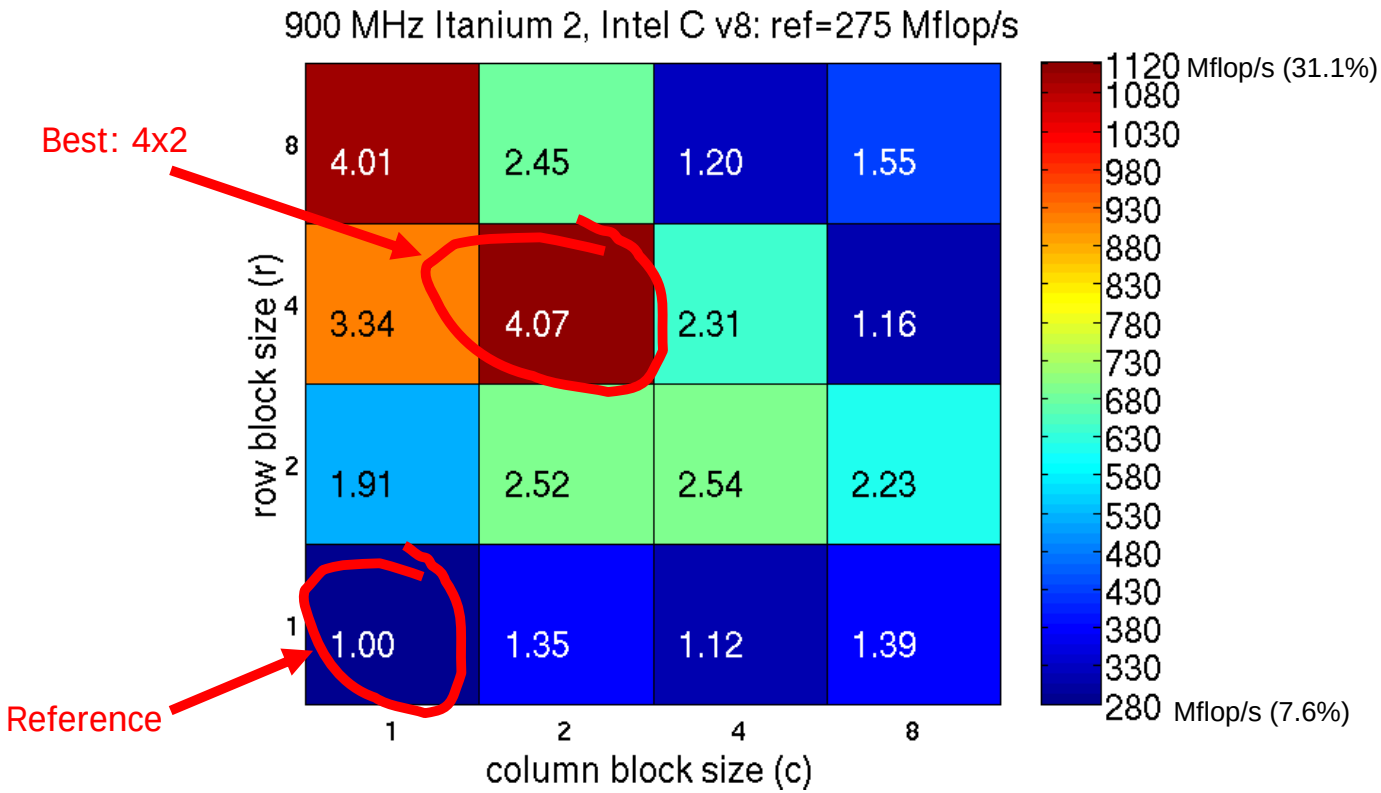
Courtesy of Jack Dongarra

Speedups on Itanium 2: The Need for Search

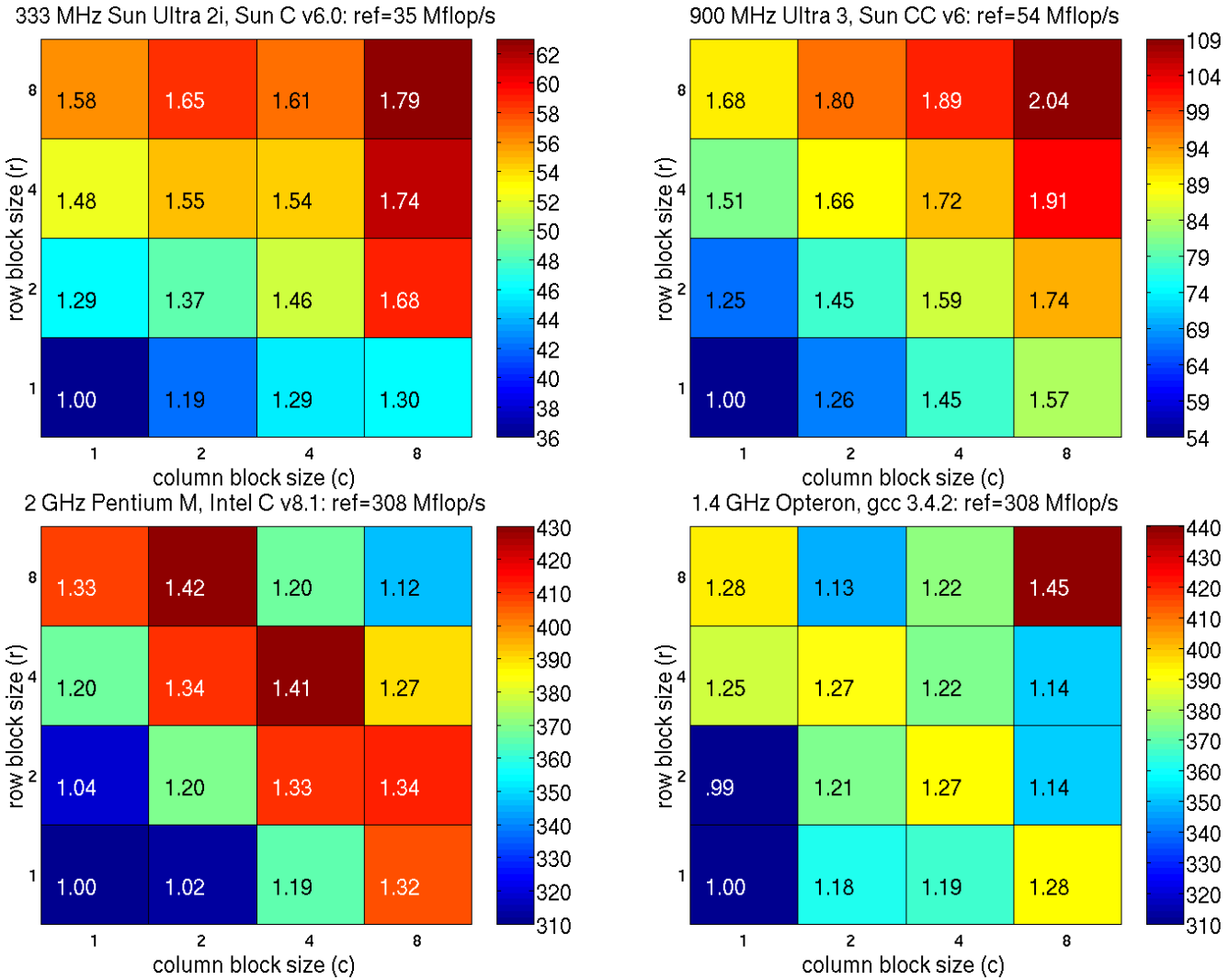


Courtesy of Jack Dongarra

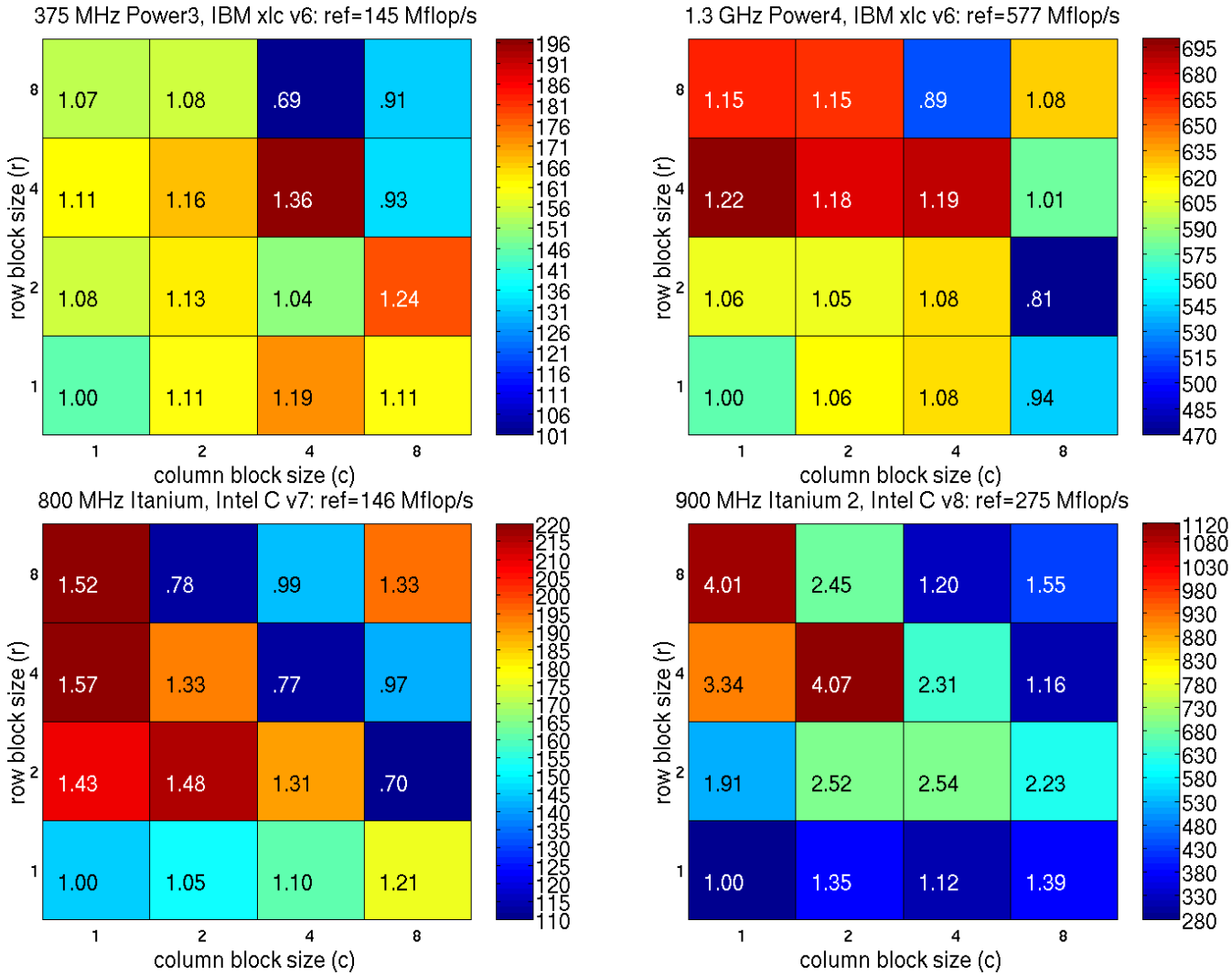
Speedups on Itanium 2: The Need for Search



Courtesy of Jack Dongarra

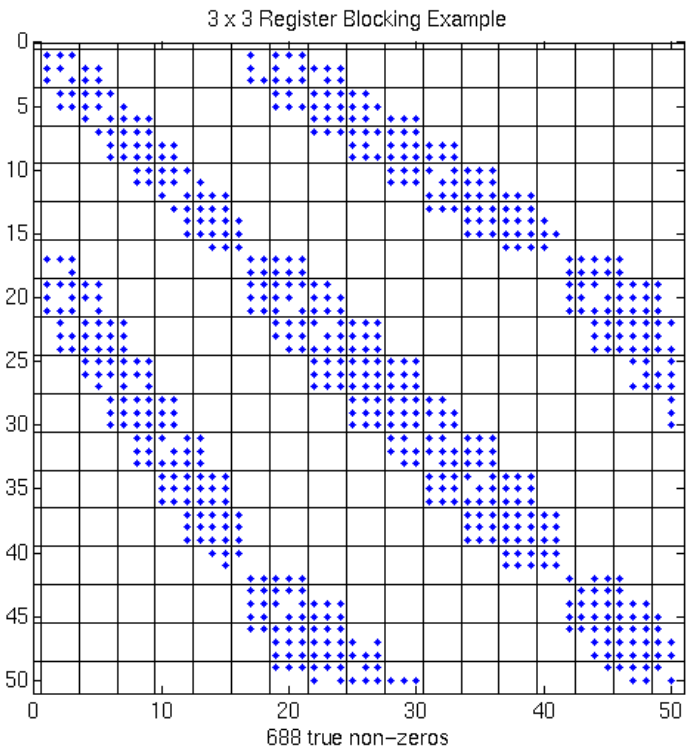


Courtesy of Jack Dongarra



Courtesy of Jack Dongarra

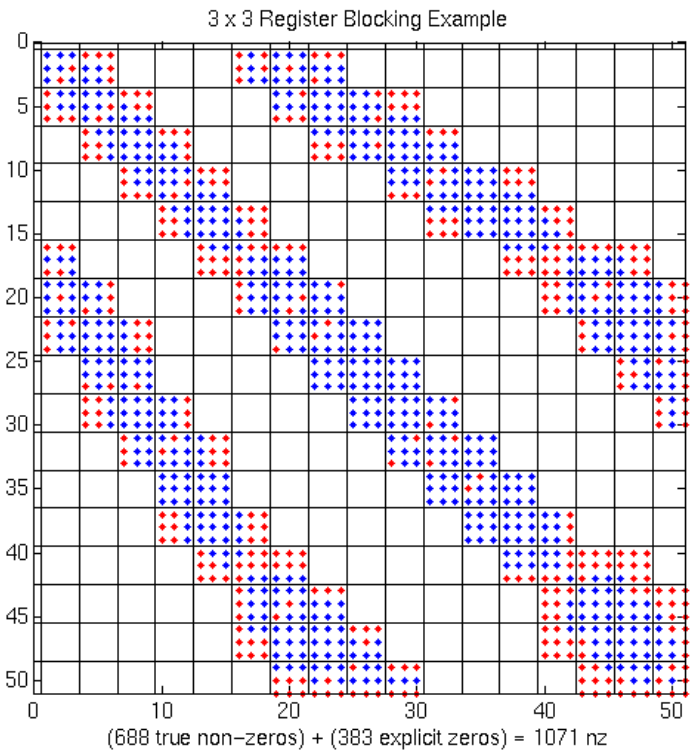
More Surprises tuning SpMV



- More complex example
- Example: 3x3 blocking
  - Logical grid of 3x3 cells

Courtesy of Jack Dongarra

Extra Work Can Improve Efficiency



- More complex example
- Example: 3x3 blocking
  - Logical grid of 3x3 cells
  - Pad with zeros
  - "Fill ratio" = 1.5
- On Pentium III:
  1.5x speedup! (2/3 time)

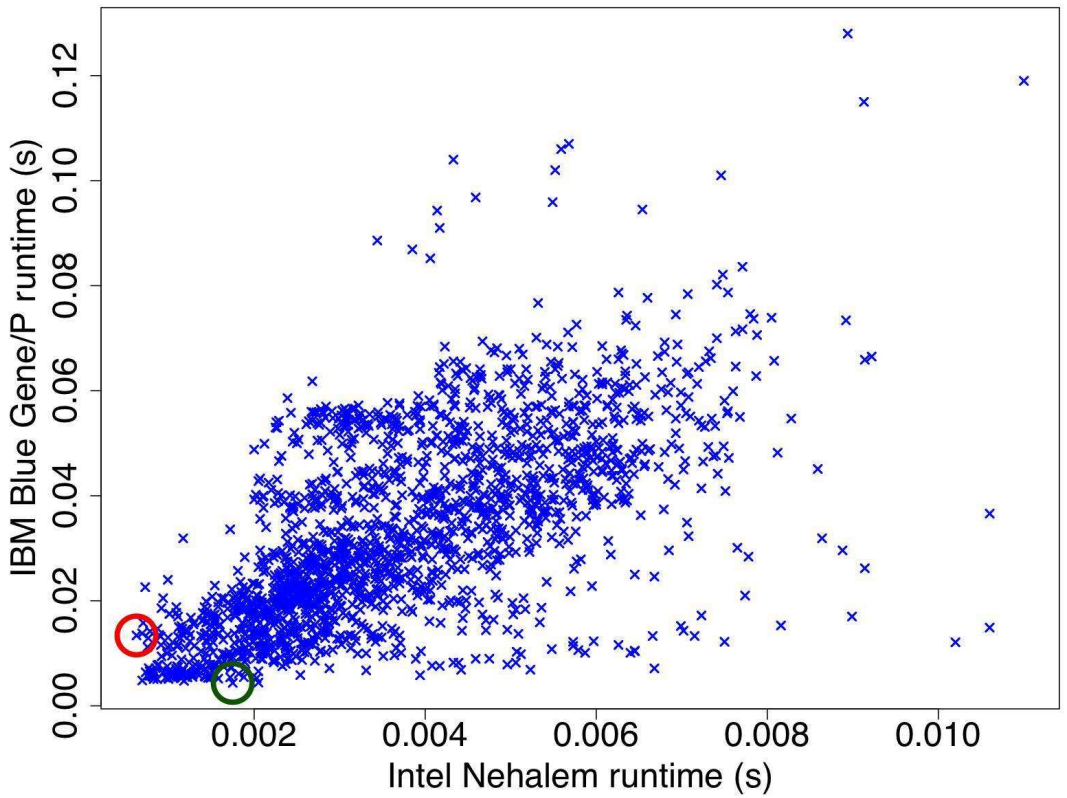
Courtesy of Jack Dongarra

How to Deal with Complexity?

- **Many parameters in the code needs to be optimized.**
- **Software adaptivity is the key for applications to effectively use available resources whose complexity is exponentially increasing**
- **Goal:**
  - **Automatically bridge the gap between the application and computers that are rapidly changing and getting more and more complex**
- **Non obvious interactions between HW/SW can effect outcome**

Courtesy of Jack Dongarra

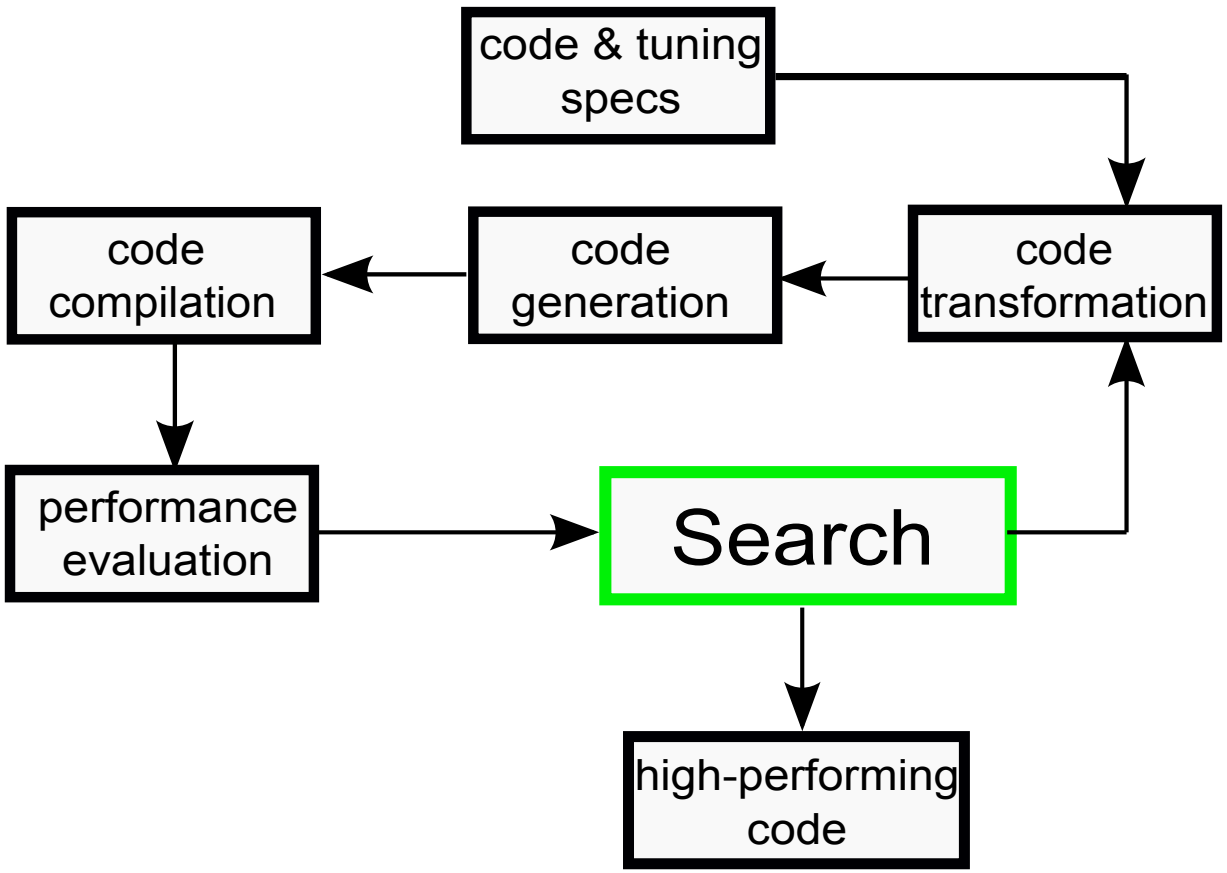
Overtuning Can Destroy Performance Portability



Each $\times$ denotes a `DGEMM` variant

Automating Empirical Performance Tuning

Given a computation kernel and transformation space:



Search in Autotuning

Alternatives:

- Complete enumeration
  - Prohibitively expensive (10^{50} variants!)
  - Unnecessary?
- Pruning
  - Careful balancing act (between aggressive and conservative strategies)

Helpful (necessary?) precursors: The expert still plays a role!

- Identify variable space (parameters to be tuned, ranges, constraints)
- Quantify measurement limitations and noise
- Incorporate known theoretical considerations (models)
- Construct meaningful objectives

→ Reduce search space and/or number of variants that need to be examined

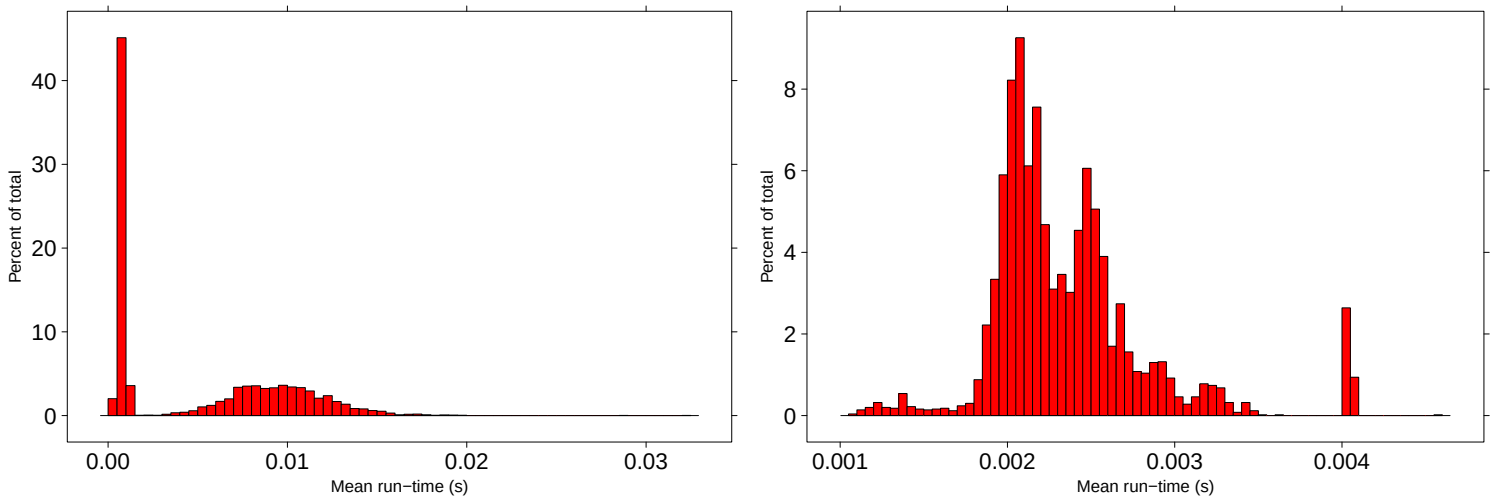
Our goal

Design, implement, and analyze *efficient optimization (=search) algorithms*
. . . for tuning kernels in small computation budgets

Is a Sophisticated Search Algorithm Needed?

[Seymour, You, & Dongarra, Cluster Computing '08]: Random search performs better than alternatives as the number of tuning parameters grows !

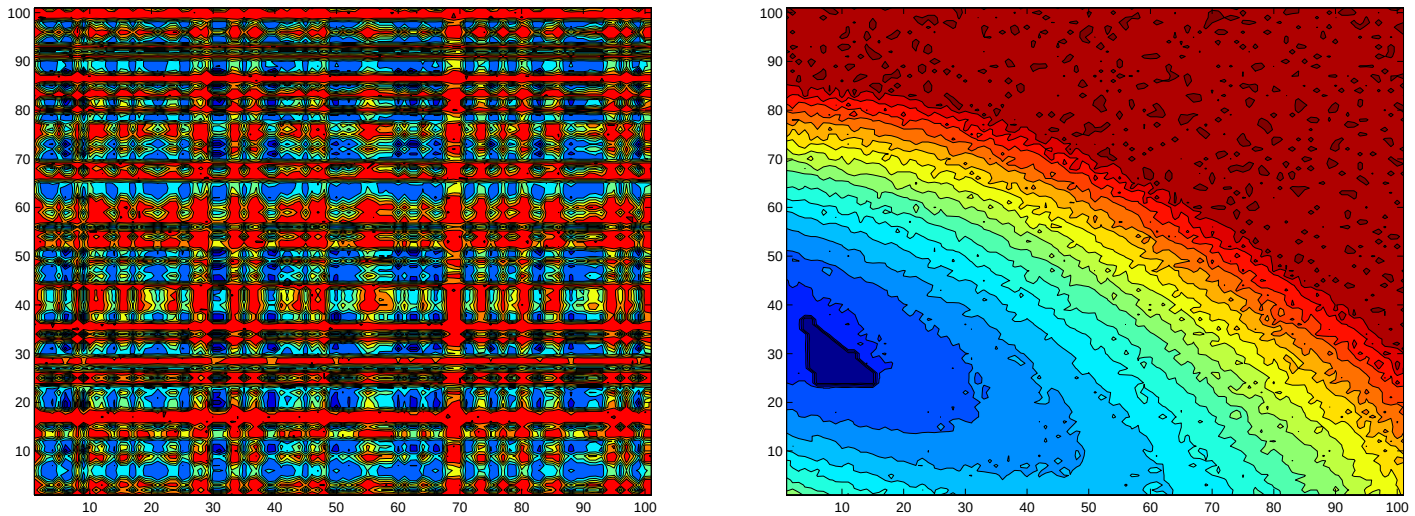
Depends on distribution of high-performing variants:



(5000 semantically equivalent variants each)

Is a Sophisticated Search Algorithm Useful?

Depends on structure of the (modeled) search space:



Both 2-dimensional problems have the same histogram

Must learn/model/exploit this structure to quickly find high-performing variants

Formulation and Modeling: Optimization is Optimization

Finding the best configuration is a mathematical optimization problem

$$\min_x \{f(x) : x = (x_{\mathcal{I}}, x_{\mathcal{B}}, x_{\mathcal{C}}) \in \mathcal{D} \subset \mathbb{R}^n\}$$

- x multidimensional parameterization (compiler type, compiler flags, unroll/tiling factors, internal tolerances, ...) for a code variant
- $f(x)$ empirical performance metric of x such as FLOPS, power, or run time (requires a run)
- $\mathcal{D}$ search domain (constraints for feasible transformation, no errors, ...)
  - bound: unroll $\in [1, \ldots, 30]$; RT $= 2^i$, i=[0,1,2,3]
  - known: $(RT_I * RT_J \leq 150)$ (cheap); power consumption ≤ 90 W (expensive)
  - hidden: transformation errors (relatively cheap), compilation (expensive), and run time (very expensive) failures

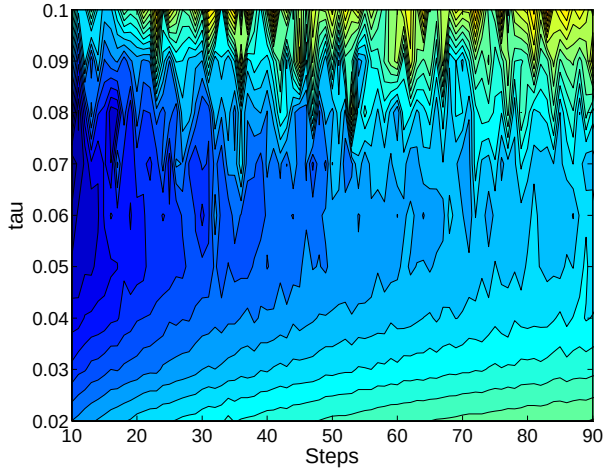
See [Balaprakash, Hovland, & W., iWAPT '11]

Optimization Challenges in Autotuning

$\min_x \{f(x) : x = (x_{\mathcal{I}}, x_{\mathcal{B}}, x_{\mathcal{C}}) \in \mathcal{D} \subset \mathbb{R}^n\}$

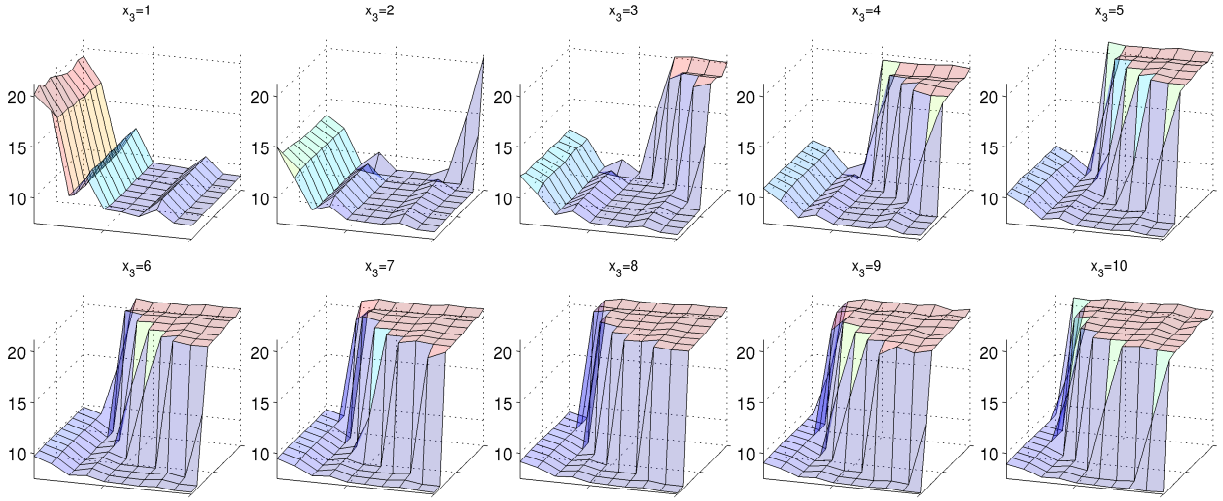
- f noisy, expensive, black box
- Discrete x unrelaxable
- $\nabla_x f$ unavailable/nonexistent
- "Cliffs", many distinct/local solutions?

Calls for **Derivative-Free Optimization**



↓ Integer Space: `MM` (MatMult)

↑ Mixed-Integer: Lattice QCD code



Courtesy of Stefan Wild

SPAPT: Orio-ready Implementation

[Norris, Hartono, & Gropp, '07]

- Extensible empirical tuning system
- Allows inserting annotations as structured comments
- Supports architecture independent and specific optimizations

```
/* AXPY Kernel */
for (i=0; i<=n-1; i++)
   y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i];
```

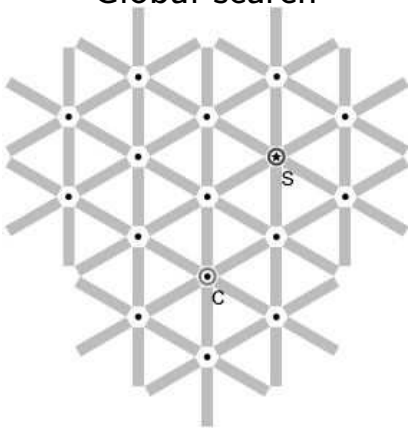
↓

/* Tuning specifications */ UF = {1,...,30}; PAR = {True, False}

```
/*@ begin Loop (
  transform Unroll(ufactor=UF, parallelize=PAR)
    for (i=0; i<=n-1; i++)
       y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i];
      )
@*/
```

Classical Algorithms for Performance Tuning

Global search



- exploration and exploitation
- find the globally best*
- long search time
- parameter sensitive

Local search

- limited exploration
- find the locally best
- short search time
- risk of bad local solution

Hypothesis: customized local search algorithms are effective for short computational budgets

Previous Algorithms for Performance Tuning

[Seymour, You, & Dongarra, Cluster Computing '08] and [Kisuki, Knijnenburg, & O'Boyle, PACT '00] compared several global and local algorithms

- Random search outperforms a genetic algorithm, simulated annealing, particle swarm, Nelder-Mead, and orthogonal search !
- Large number of high-performing parameter configurations → easy to find one of them

[Norris, Hartono, & Gropp, *Computational Science* '07] used several global and local algorithms but no comparison

- Nelder-Mead simplex method, simulated annealing, a genetic algorithm

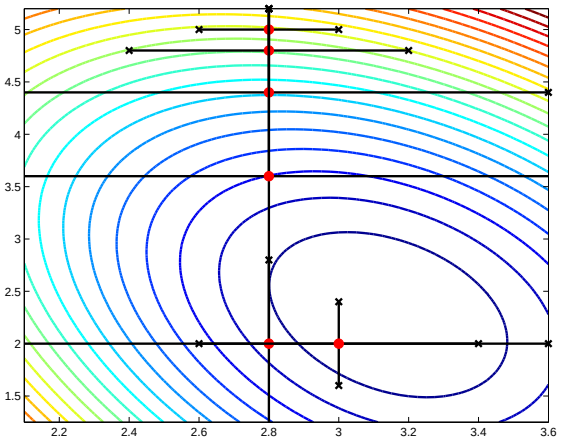
Other local search algorithms without comparison to global search:

- Orthogonal search in ATLAS [Whaley & Dongarra, SC '98]
- Pattern search in loop optimization [Qasem, Kennedy, & Mellor-Crummey SC '06]
- Modified Nelder-Mead simplex algorithm in Active Harmony [Tiwari, Chen, Chame, Hall, & Hollingsworth, IPDPS '09]

Local Algorithms: Direct Search Methods

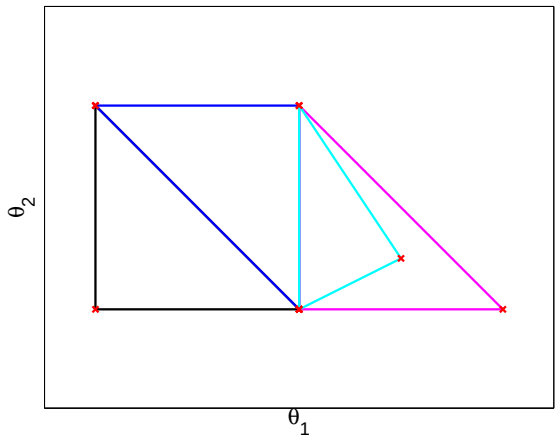
See [Kolda, Lewis, & Torczon, *SIREV* '03]

Pattern Search



Easy to parallelize f evaluations

Nelder-Mead

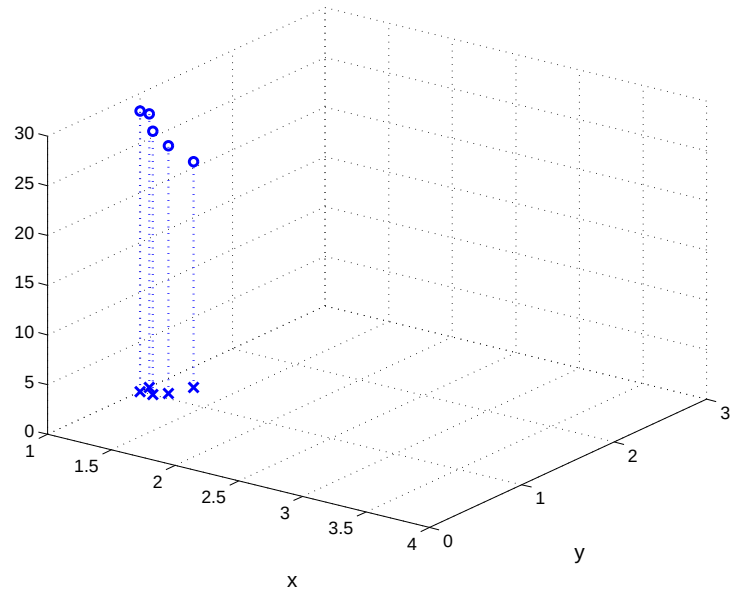


Popularized by *Numerical Recipes*

- Rely on indicator functions: $[f(x_k + s) <^{?} f(x_k)]$
  - Ignore valuable information on relative magnitudes of $f(x_k)$

Making the Most of Little Information on f

- f is expensive $\Rightarrow$ can afford to make better use of points
- Overhead of the optimization routine is minimal (negligible?) relative to cost of empirical evaluation



Bank of data, $\{x_i, f(x_i)\}_{i=1}^{k}$:

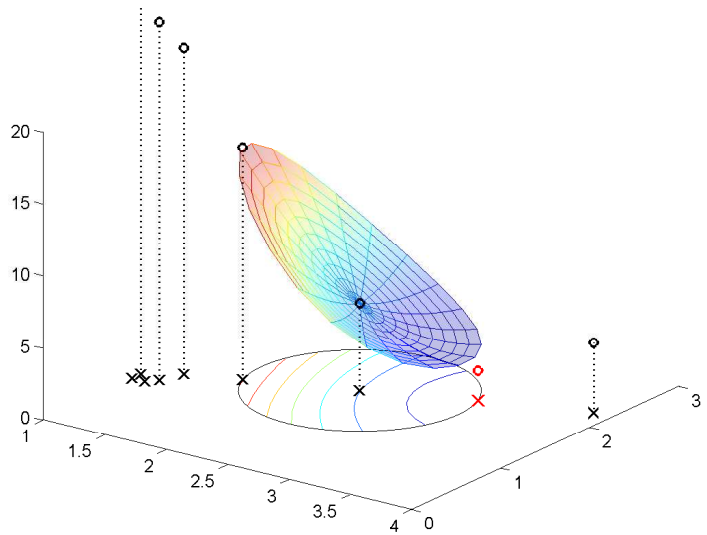
$=$ Everything* known about f

Idea:

- Make use of growing bank as optimization progresses

Making the Most of Little Information on f

- f is expensive $\Rightarrow$ can afford to make better use of points
- Overhead of the optimization routine is minimal (negligible?) relative to cost of empirical evaluation



Bank of data, $\{x_i, f(x_i)\}_{i=1}^{k}$:

$=$ Everything* known about f

Idea:

- Make use of growing bank as optimization progresses

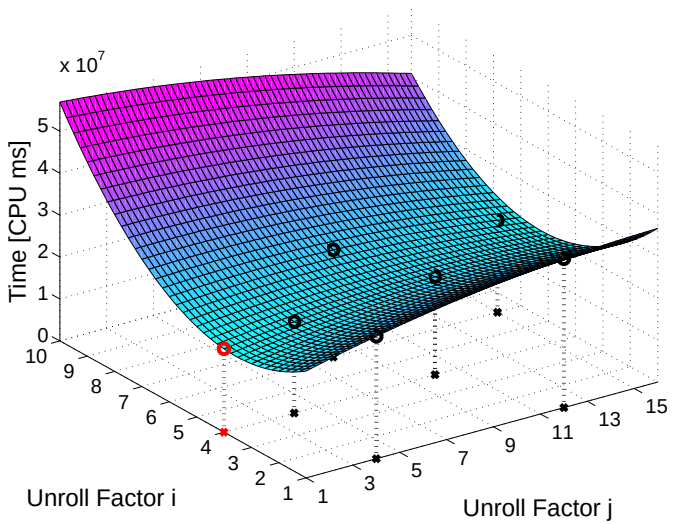
Surrogate-Based Trust-Region Algorithms

Substitute $\min \{m(x) : x \in \mathcal{B}_k\}$ for $\min f(x)$

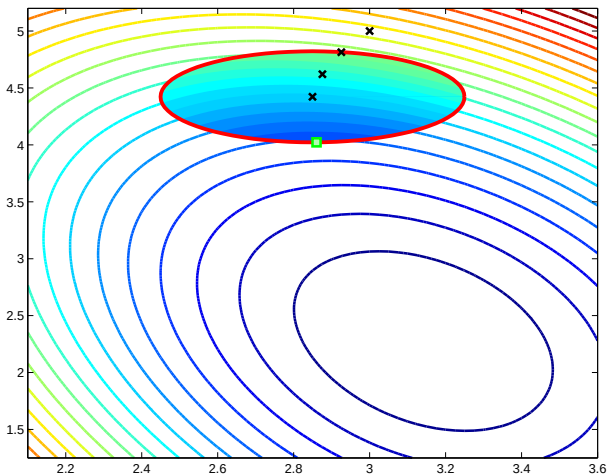
f expensive, no ∇f

m cheap, analytic derivatives

Surrogate based on known f values



Trust $m \approx f$ in $\mathcal{B}_k$

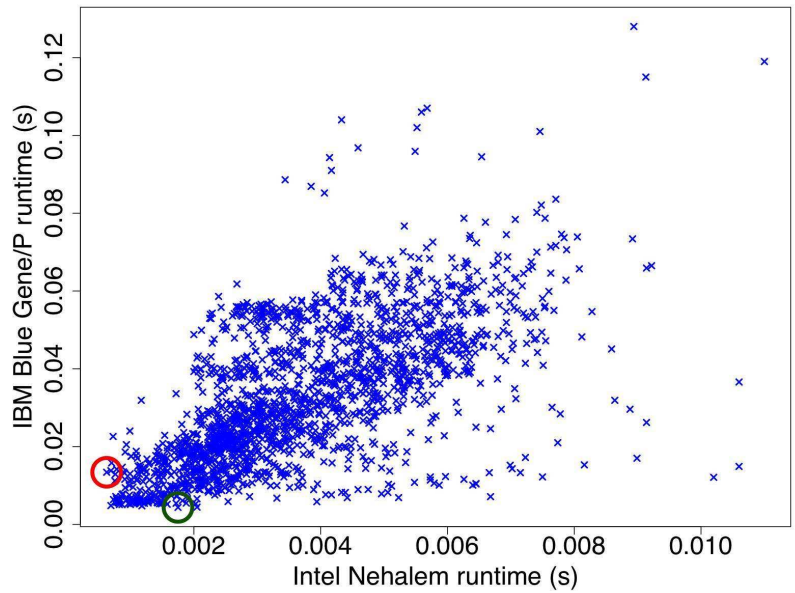


Surrogates: predict improvement

Simultaneously Optimizing Multiple Objectives

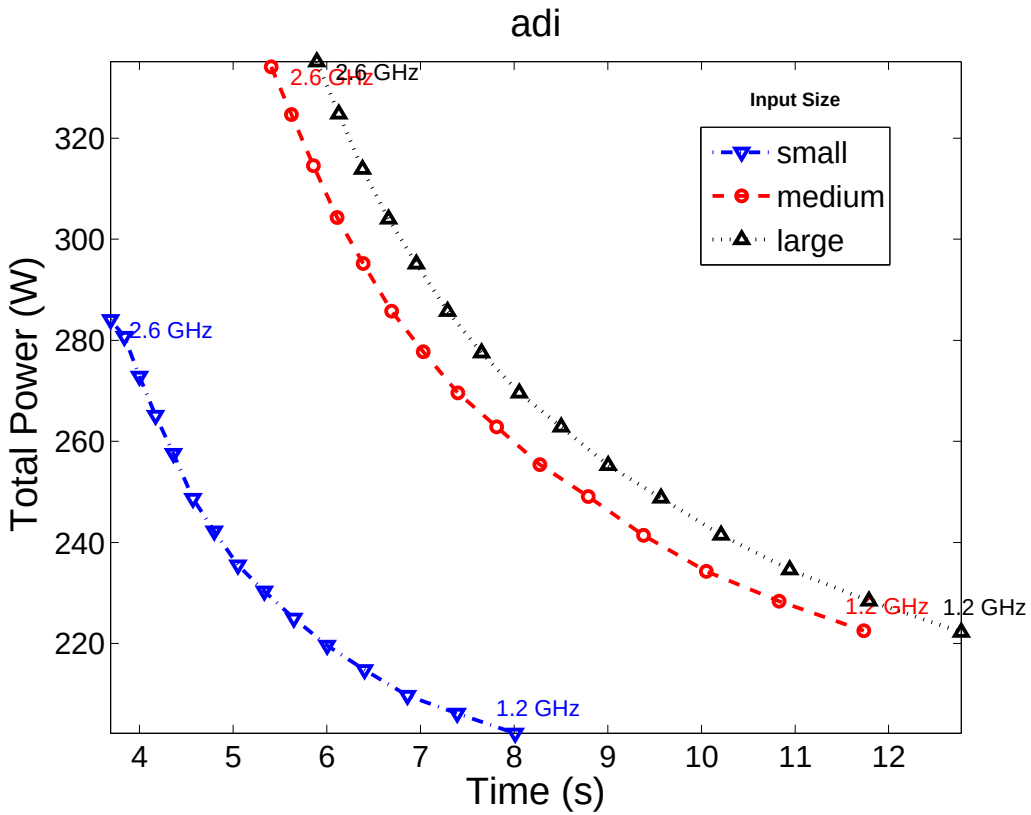
$$\min_{x \in \mathcal{D}} \{f_1(x), f_2(x), \ldots, f_p(x)\}$$

- No *a priori* weights w_i $\left(\sum_i w_i f_i(x)\right)$
- Dominated points $\tilde{x}$: $\exists x^* \in \mathcal{D}$ with $f_i(\tilde{x}) \geq f_i(x^*)\ \forall i$, $f_j(\tilde{x}) > f_j(x^*)$ some j
- Seek Pareto front of non-dominated points



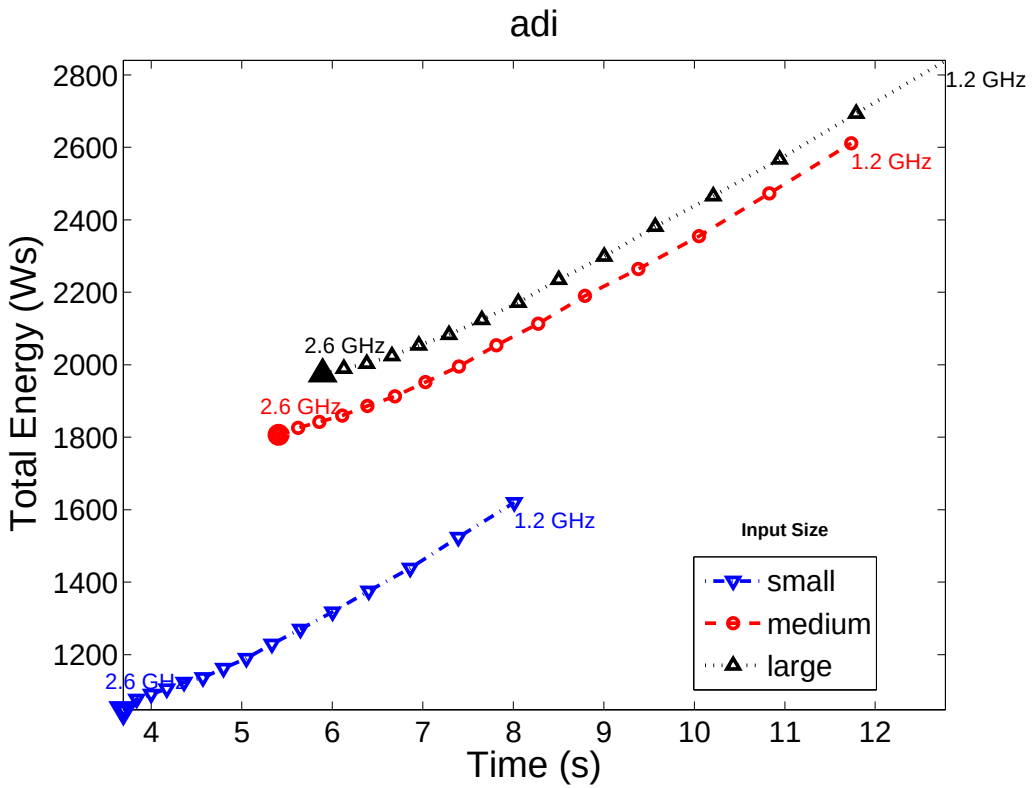
Courtesy of Stefan Wild

Multiple Objectives: Time, Power, Energy



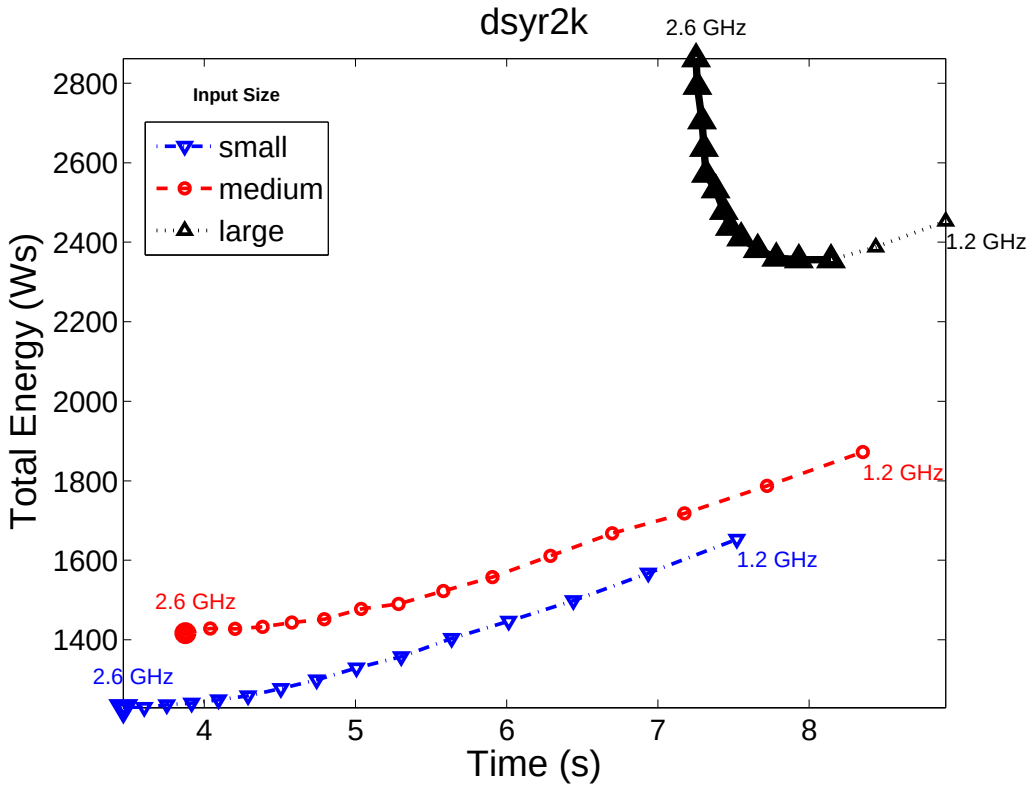
- Tradeoffs in power do not imply tradeoffs in energy

Multiple Objectives: Time, Power, Energy



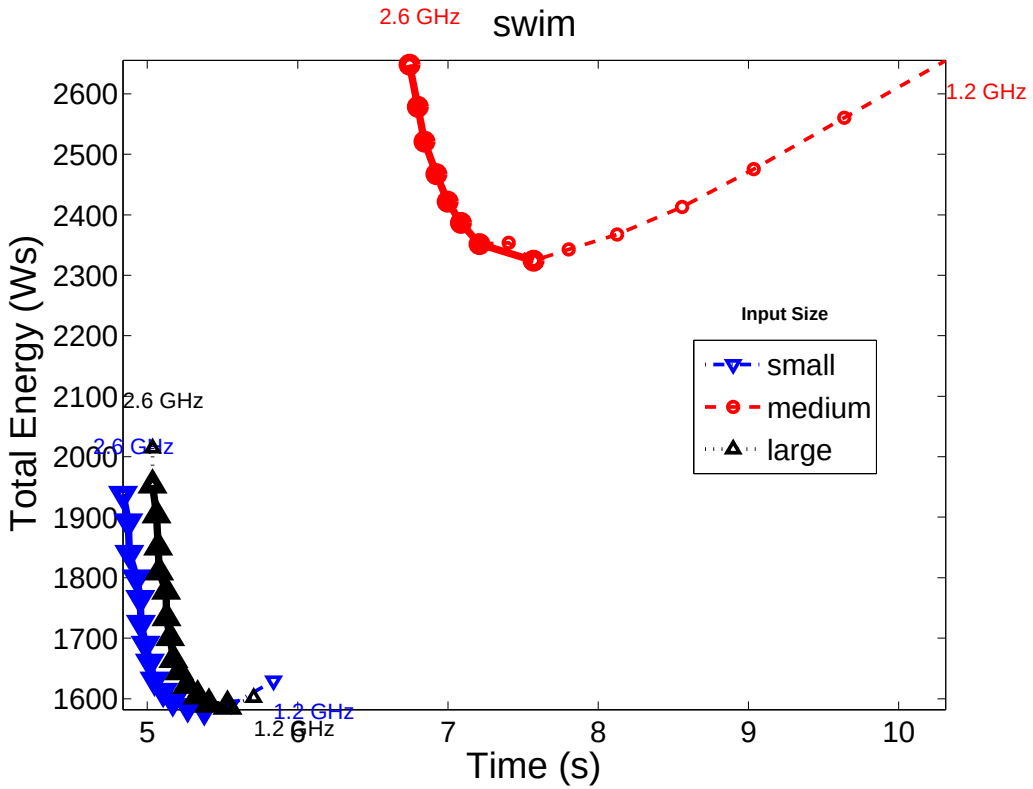
- Tradeoffs in power do not imply tradeoffs in energy
- Objectives may not be conflicting: "Race to idle

Multiple Objectives: Time, Power, Energy



- Tradeoffs in power do not imply tradeoffs in energy
- Objectives may not be conflicting: "Race to idle
- Tradeoffs occur for different sizes

Multiple Objectives: Time, Power, Energy



- Tradeoffs in power do not imply tradeoffs in energy
- Objectives may not be conflicting: "Race to idle
- Tradeoffs occur for different sizes
- Tradeoffs occur at different frequencies

Summary and Links

- Performance tuning increasingly necessary, not yet "automatic"
- Derivative-free optimization is a powerful, practical tool

When the available tuning time is limited:

- Global exploration less useful
- Problem formulation and starting point play important roles

Future work includes:

- Incorporation of models, binary parameters, constraints (from models or otherwise), online restart strategies,role in full application codes, ...

→ always collecting new search/optimization problems
... especially those with structure

Some preprints `http://mcs.anl.gov/~wild`

Orio `http://trac.mcs.anl.gov/projects/performance/wiki/Orio`

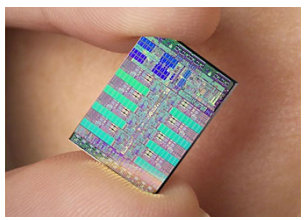
SPAPT `http://trac ··· /performance/browser/orio/testsuite/SPAPT.v.01`

Outline

1. Communication "Avoiding" Algorithms
   - Cache oblivious algorithm
   - Parallel Algorithm
2. Synchronization-reducing algorithms
   - DAG generation
   - DAG generation
   - Granularity and Hybrid Computing
3. Auto-tuning
4. Reproducibility and Mixed-Precision Methods

Iterative Refinement: for speed

- **What if double precision much slower than single?**
  - **Cell processor in Playstation 3**
    - **256 GFlops single, 25 GFlops double**



  - **Pentium SSE2: single twice as fast as double**
- **Given Ax=b in double precision**
  - **Factor in single, do refinement in double**
  - **If $\kappa(A) < 1/\varepsilon_{single}$, runs at speed of single**
- **1.9x speedup on Intel-based laptop**
- **Applies to many algorithms, if difference large**

Courtesy of Jack Dongarra

Reproducibility

- Reproducible numerical computations is already difficult for a simple reduce.
- The increase of PUs, dynamic scheduling and the use of hybrid mixed-precision hardware makes it even harder.
- Changing algorithms may be particularly harmful.

Fast Matrix Multiplication (1)

(Cohn, Kleinberg, Szegedy, Umans)

- Can think of fast convolution of polynomials p, q as
  - Map p (q) into group algebra $\Sigma_i p_i z^i \in \mathbf{C}[G]$ of cyclic group $G = \{ z^i \}$
  - Multiply elements of $\mathbf{C}[G]$ (use divide&conquer = FFT)
  - Extract coefficients
- For matrix multiply, need non-abelian group satisfying triple product property
  - There are subsets X, Y, Z of G where xyz = 1 with $x \in X, y \in Y, z \in Z \Rightarrow x = y = z = 1$
  - Map matrix A into group algebra via $\Sigma_{\mathbf{xy}} A_{\mathbf{xy}} x^{-1}y$, B into $\Sigma_{\mathbf{y'z}} B_{\mathbf{y'z}} y'^{-1}z$.
  - Since $x^{-1}y\, y'^{-1}z = x^{-1}z$ iff y = y' we get $\Sigma_{\mathbf{y}} A_{\mathbf{xy}} B_{\mathbf{yz}} = (AB)_{\mathbf{xz}}$
- Search for fast algorithms reduced to search for groups with certain properties
  - Fastest algorithm so far is $O(n^{\mathbf{2.38}})$, same as Coppersmith/Winograd

Courtesy of Jack Dongarra

MPI. Really ?

- Hybrid parallelism (MPI+openMP) is tricky.
- MPI 3.0 introduces among other things neihborhood collective communications, asynchronous collective operations, the ability to hint the middleware about possible optimizations, ...
- MPI 3.0still considers MPI ranks as process and not as "end-points". :(
- MPI will have trouble going to exascale. Another approach is to resort to data parallel languages to express data parallelism. HPF removed power from power users compared to MPI, which is one of the reason for the success of MPI.

If you are wondering what's beyond ExaFlops

Mega, Giga, Tera, Peta, Exa, Zetta ...

10^{3}	kilo
10^{6}	mega
10^{9}	giga
10^{12}	tera
10^{15}	peta
10^{18}	exa
10^{21}	zetta

10^{24}	yotta
10^{27}	xona
10^{30}	weka
10^{33}	vunda
10^{36}	uda
10^{39}	treda
10^{42}	sorta
10^{45}	rinta
10^{48}	quexa
10^{51}	pepta
10^{54}	ocha
10^{57}	nenaN
10^{60}	minga
10^{63}	luma

50

Courtesy of Jack Dongarra