Programming for Parallelism and Locality with Hierarchically Tiled Arrays

Draft Version 2.0

Ganesh Bikshandi, Jia Guo, Dan Hoeflinger, Maria Garzaran, Basilio Fraguela, Christoph von Praun, Gheorghe Almasi, David Padua

Problem Statement

- To design & develop language extensions for *parallelism*
- To design & develop language extensions for *locality*
- To quantify the Performance loss due to *abstraction*

Motivation

- Memory Hierarchy is evolving
  - **Registers - Cache (L1, L2) - Disk - Distributed Memory**
- Programming the future systems is a key challenge
  - **Program Portability, Performance Portability, Programmer Productivity**
- Automatic compiler-assisted program transformations are limited
  - **Compiler analysis is not accurate**

State of the art

- What is wrong with MPI?
  - $low-level \implies lesser\ productivity$
  - $SPMD \implies unstructured\ codes \implies bugs$
- What is wrong with Languages?
  - $language \implies compiler \implies complexity$
  - $learning \implies development\ cost$
  - $specific \implies versioning \implies maintenance\ cost$

Properties

- Required
  - **Global view**
  - **Single threaded**
  - **Explicit performance control**
- Desired
  - **Minmal changes to existing sequential code**
  - **Minimal set of operations**

- ***TODO: give a specific example for each case???***
- ***for e.g. illustrate why global view is preferred???***

Overview of our Approach

- Introduce a new Data Type for tiling
  Hierarchically Tiled Arrays
  - **Recursive data structure**
  - **A tree-structured representation of memory**
- Introduce new operations
  operator overloading
  - **High-level, F90-like operations**
  - **Transparent implementation**
  - **Semantics Preserving**
  - **Fine granularity**
- Extend MATLAB and C++
  - **MATLAB is a dynamically typed, array based, interpreted language**
  - **C++ is a statically typed, scalar/object based, compiled language**

Rationale

- Tiling is *common* to parallelism and locality
  - **Tiling the loops for locality**
  - **Tiling the data (array) for distribution**

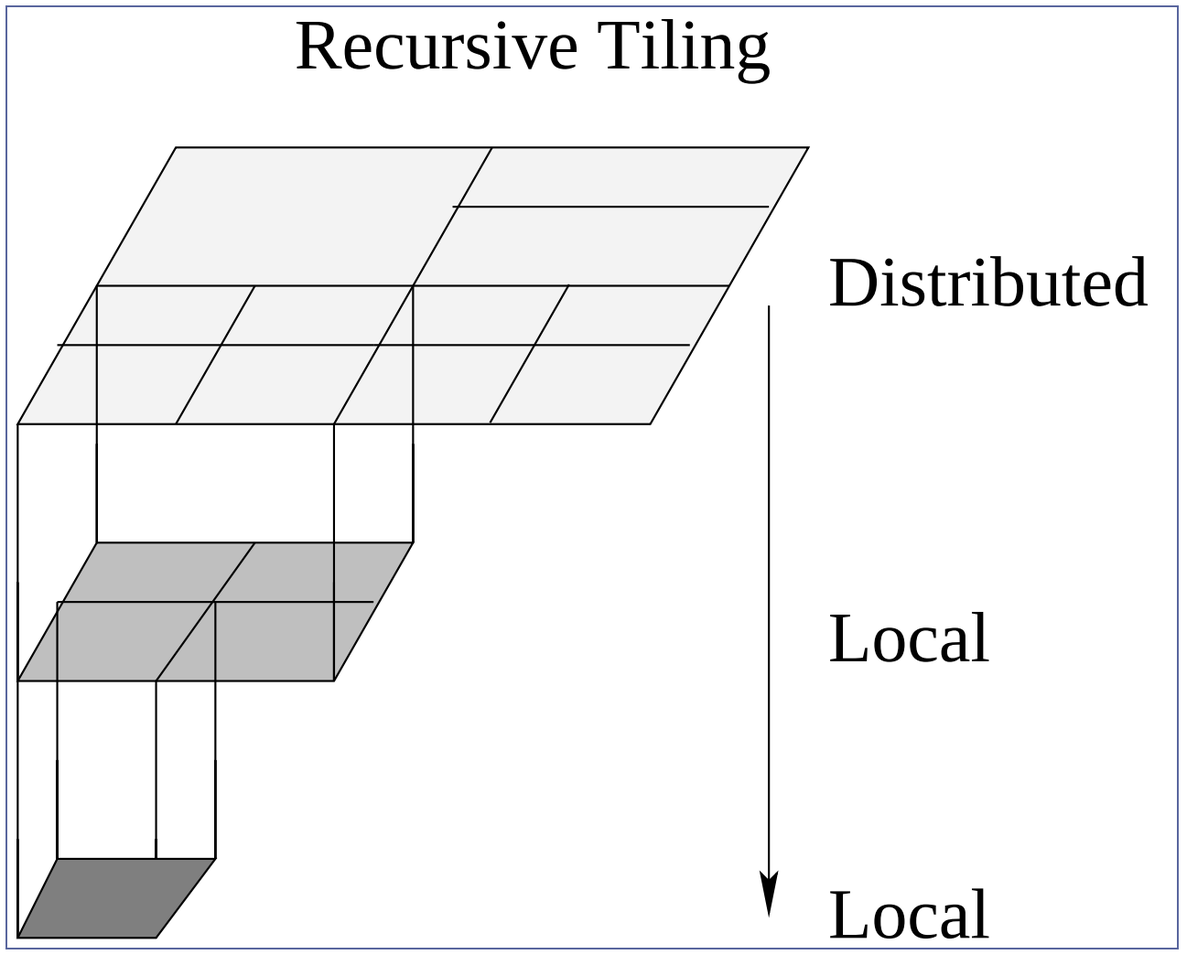
- Tiling *benefits* parallelism and locality
  - **Tiling aggregates messages in Message Passing Machines**
  - **Tiling reduces bus contention in SMP**
  - **Tiling improves locality in a single processor**

Hierarchically Tiled Arrays

- Type + Shape = Array
  - **Layout = Column or Row Major**
- Type + Shape + Tiling + [Mapping] = HTA
  - **Layout = Mapping $\odot$ Column Major $\odot$ Row Major $\odot$ Tile Major**
  - $tiling_0 = \perp$. $tiling_k = f(tiling_{k-1})$

Hierarchically Tiled Arrays

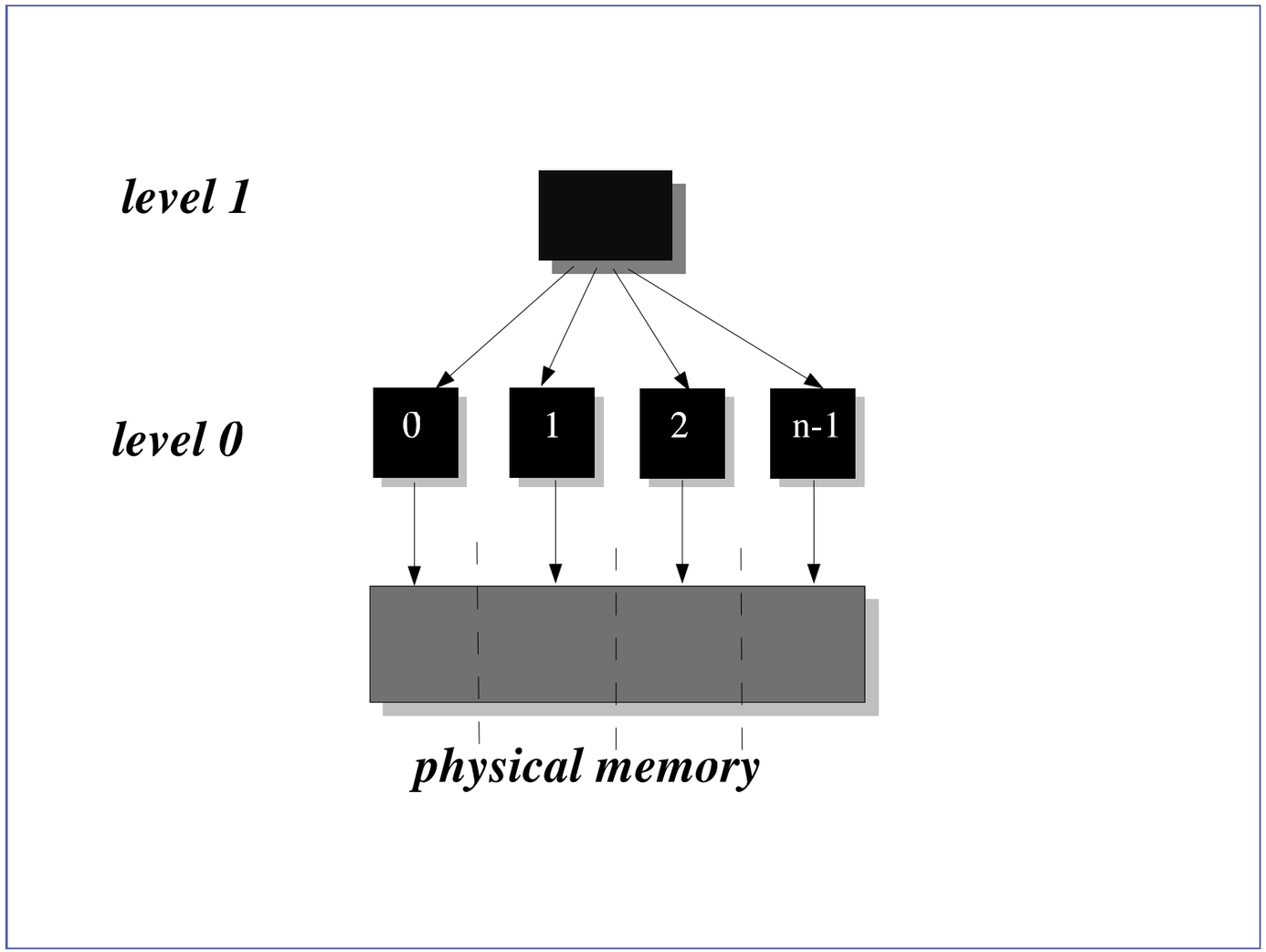
Pictorial View



- *TODO: mark Processors, Cache etc*

Hierarchically Tiled Arrays

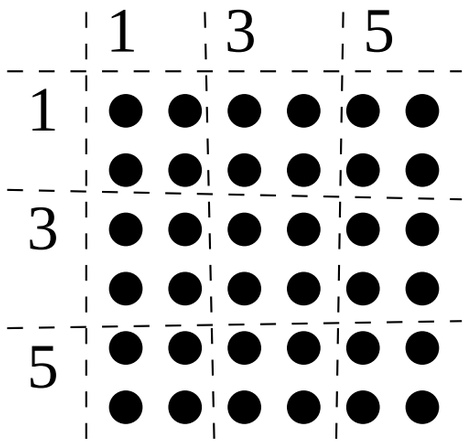
Internal Representation



- TODO: It is not clear what is the significance of this picture???

HTA Construction

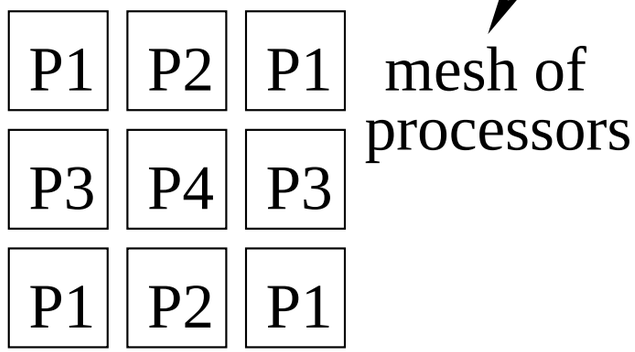
F= hta(M, {[1 3 5],[1 3 5]})



matrix M

(a)

F= hta(M, {[1 3 5],[1 3 5]}, [2,2])

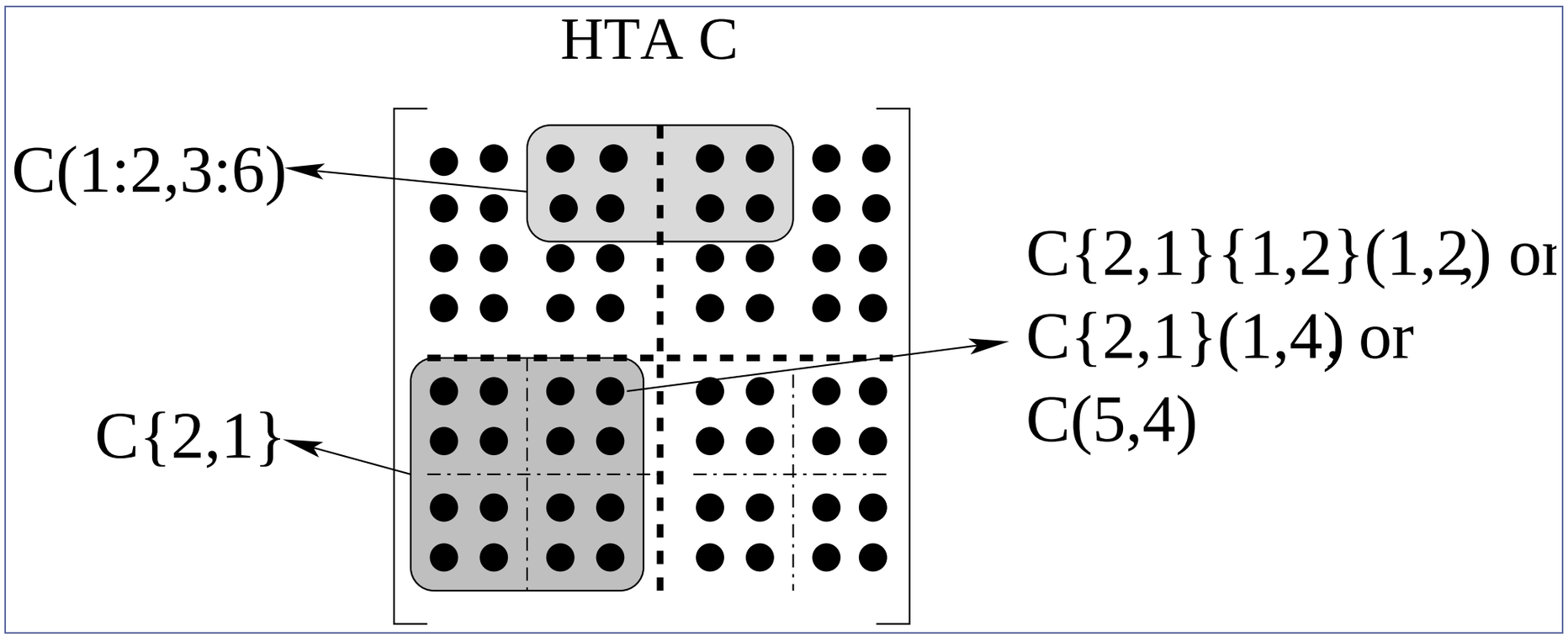


mesh of processors

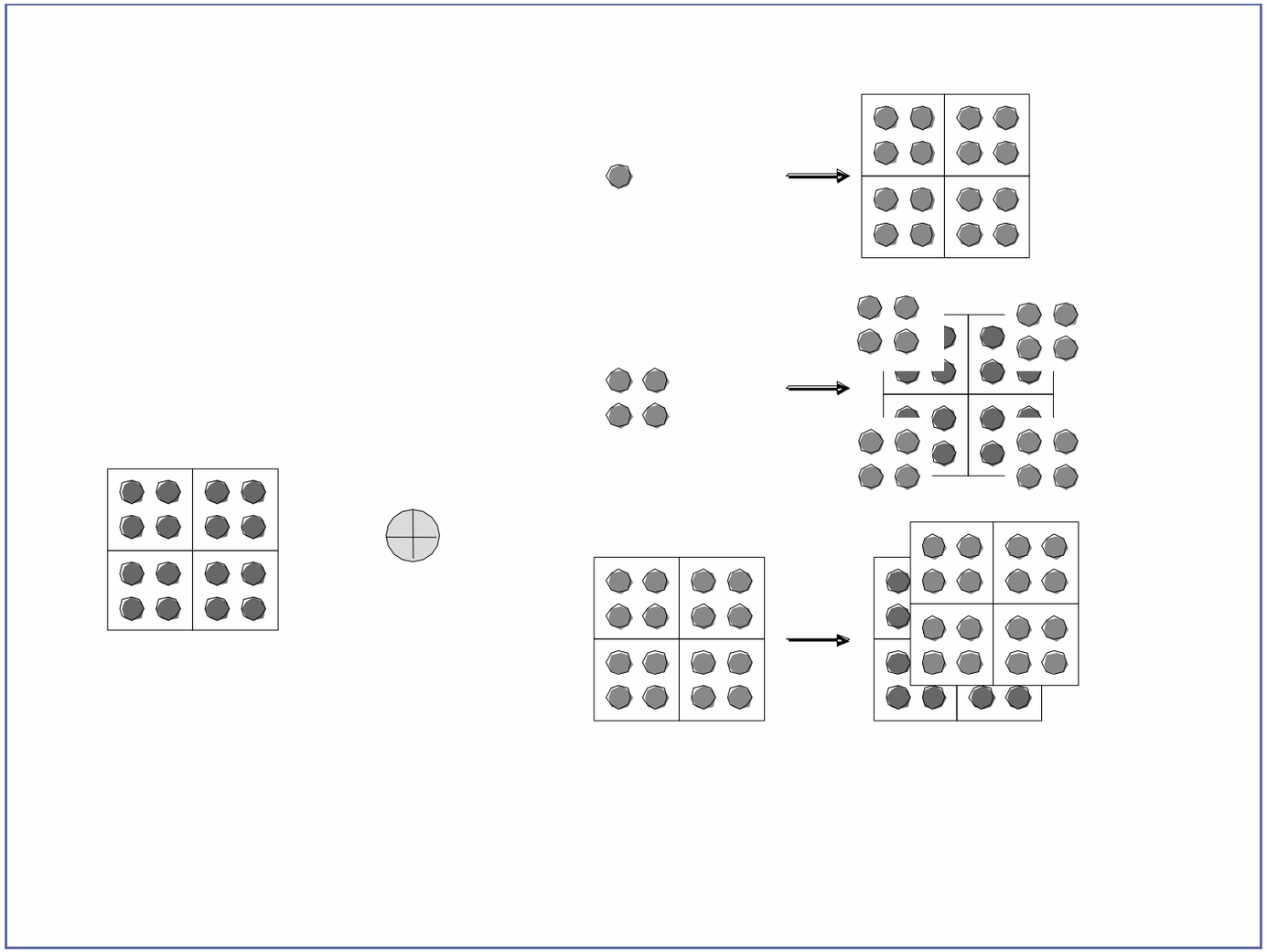
distributed HTA F

(b)

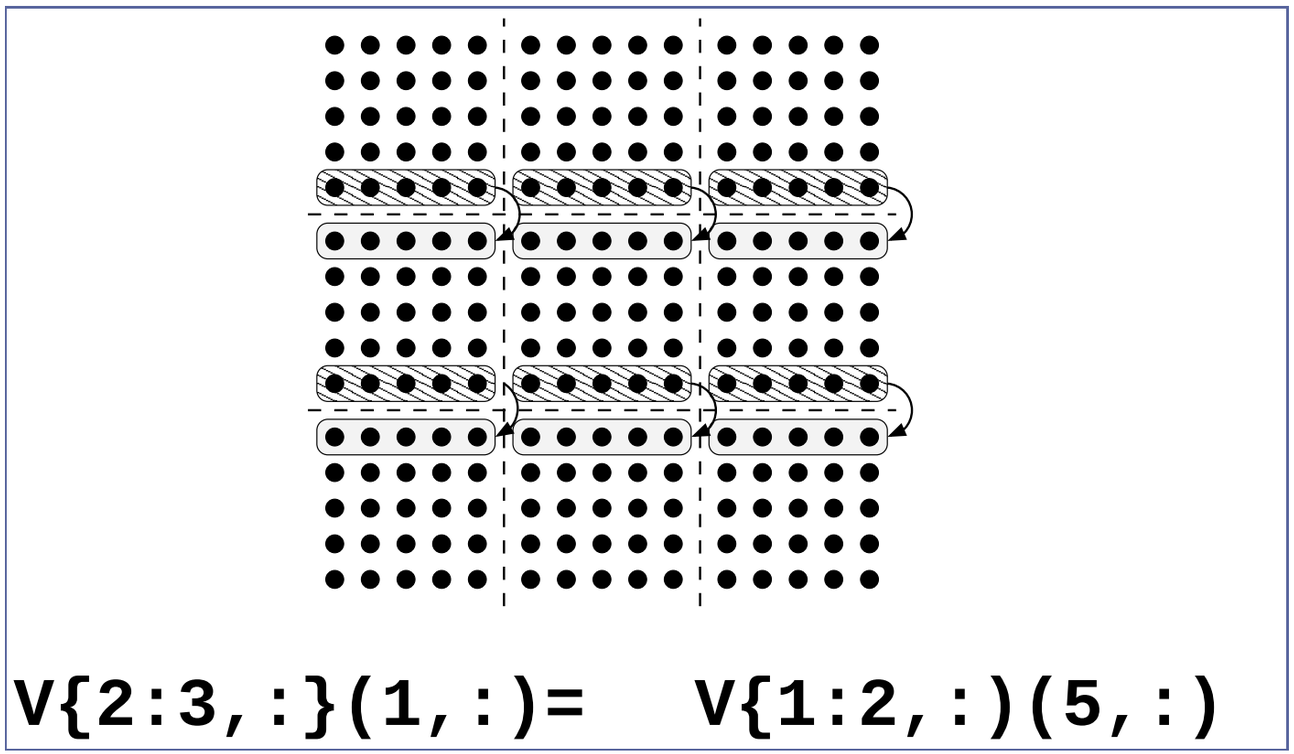
HTA Indexing



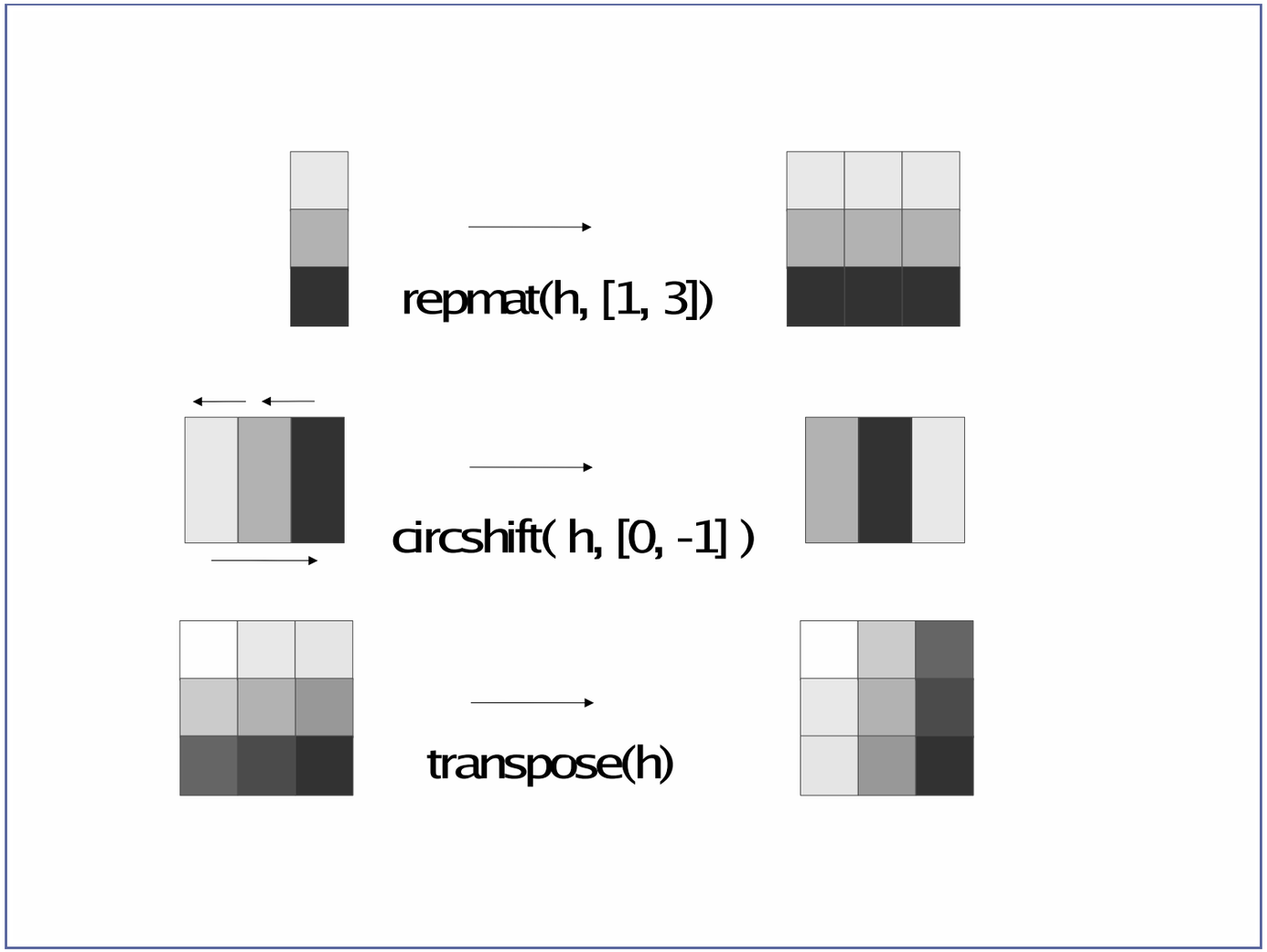
HTA Operations



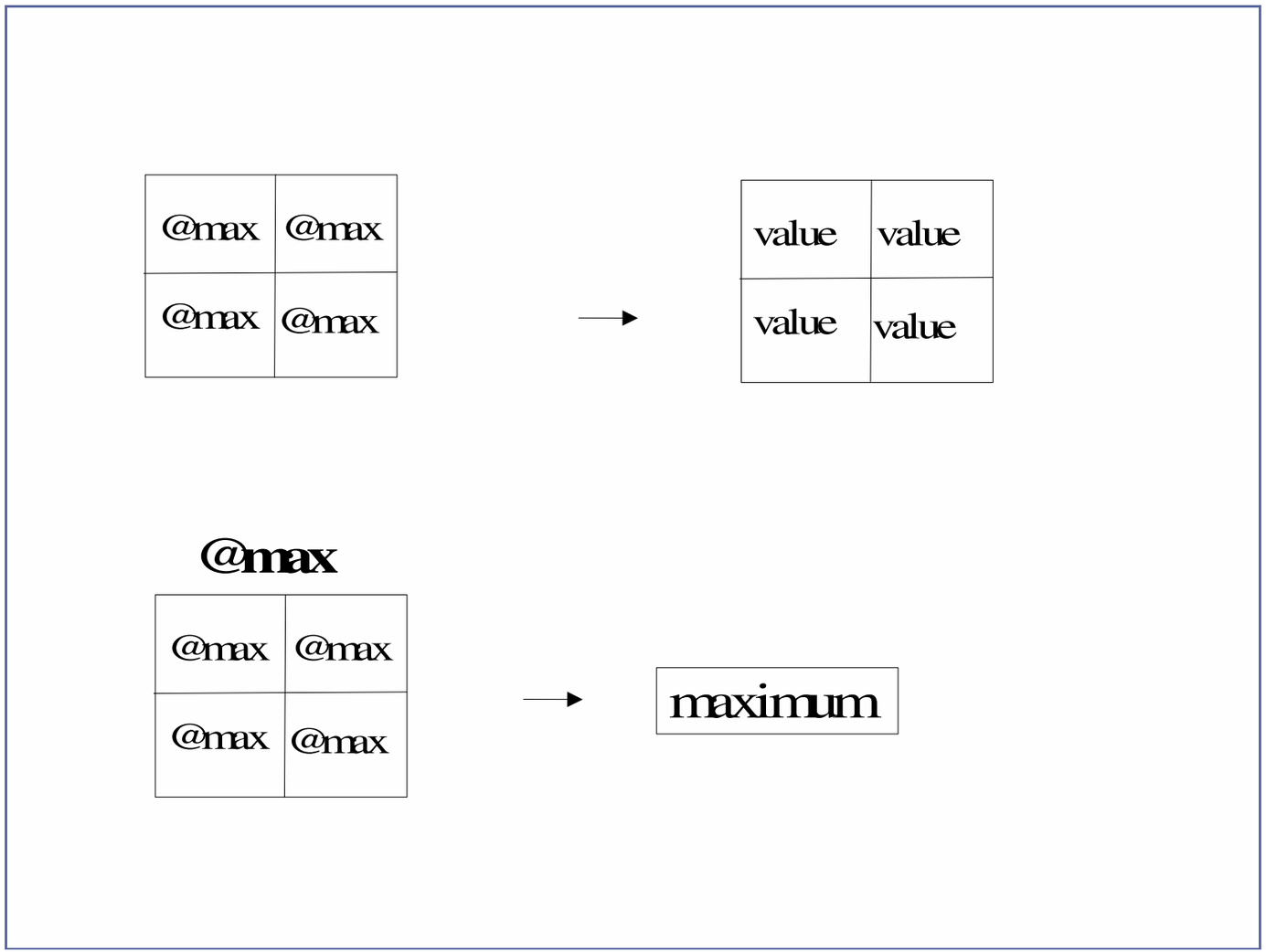
HTA Assignment



Higher Level Operations



Map/Reduce


@max
@max
@max
@max
value
value
value
value
@max
@max
@max
@max
@max
maximum

SUMMA Matrix Multiplication

Pseudocode

- Using Rank-1 Update

SUMMA-MATMUL (A, B, C)

1. $C_{ij} = 0$
2. **for** k $\leftarrow$ 0 to m-1
3. broadcast A_i^k within my row
4. broadcast B_k^j within my row
5. $C_{ij} = C_{ij} + A_i^k * B_k^j$;

SUMMA Matrix Multiplication

MATLAB

- Using Matrix Mulitplication

```
function C = summa (A, B, C)
  for k=1:m
     T1 = repmat(A{:, k}, 1, m);
     T2 = repmat(B{k, :}, m, 1);
     C = C + T1 * T2;
  end
```

Recursive Blocked Matrix Matrix Multiplication

Pseudocode

```
REC-BLOCK-MULT (A, B, C)
1.  if (size(A) = 1)
2.                  then C = C + A * B;
3.  else
4.                  for i ← 1 to size(A, 1)
5.                                  for j ← 1 to size(B, 1)
6.                                                  for k ← 1 to size(B, 2)
7.                                                                  REC-BLOCK-MULT (A_ik, B_kj, C_ij)
```

Recursive Matrix Multiplication

C++

```
template <int LEVEL> void mult(
                    HTA<double, 2, LEVEL> A,
                    HTA<double, 2, LEVEL> B,
                    HTA<double, 2, LEVEL> C) {
  int M = A.shape()[0].size();
  int N = B.shape()[0].size();
  int Q = B.shape()[1].size();
  for (int i = 0; i< M; i++) {
    for (int k = 0; k < N; k++) {
        for (int j = 0; j< Q; j++) {
          mult (A[T(i,k)], B[T(k,j)], C[T(i,j)]);
  }}}}

void mult(double& A,double& B,double& C)
{
  C += A * B;
}
```

NAS Benchmarks

Complex communication patterns

inner loop parallelism

- **Embarrassingly Parallel (EP)**
- **Neighbor Communication (MG)**
- **Complex transposes (FT)**
- **Frequent Reductions (CG)**
- **All to All personalized Communication (IS)**
- **Linear Recurrence (LU)**
- **Alternating Communication Phases (BT)**

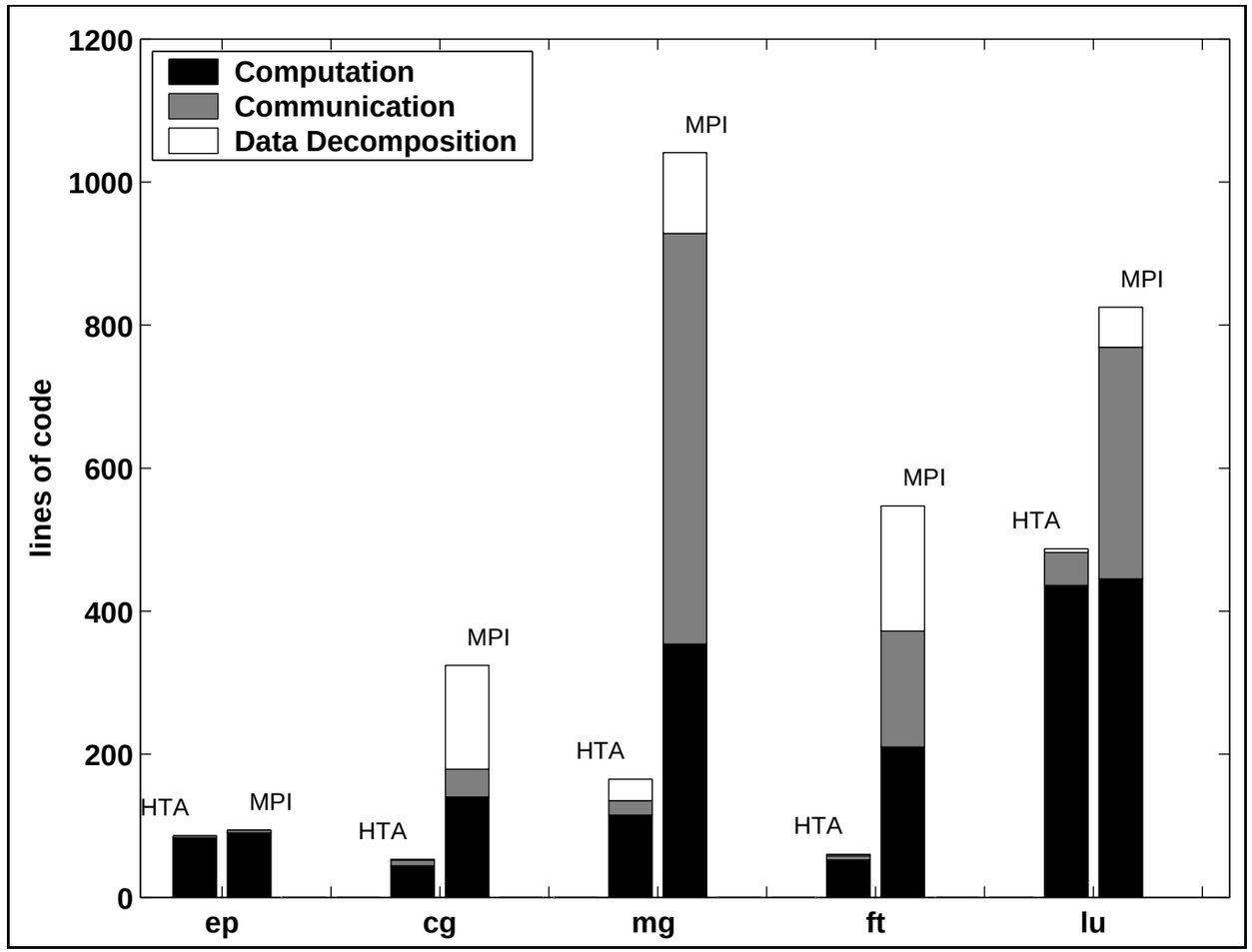
Experimental Results

MATLAB - Parallelism

- Create Serial and Parallel (HTA) versions
- Hand Vectorize the codes
- Obtain speedup relative to the serial version
- Compare with hand-written optimized F77+MPI (NPB)
- 3.2 GHz Intel Xeon, Gigabit Ethernet

Nprocs	EP (CLASS C)		FT (CLASS B)		CG (CLASS C)		MG (CLASS B)		LU (CLASS B)	
	Fortran+ MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA
1	901.6	3556.9	136.8	657.4	3606.9	3812.0	26.9	828.0	15.7	245.1
4	273.1	888.8	109.1	274.0	362.0	1750.9	17.0	273.8	6.3	60.5
8	136.3	447.0	65.5	159.3	123.4	823.6	9.6	151.3	2.9	29.9
16	68.6	224.8	37.2	87.2	89.5	375.2	4.8	87.0	1.2	16.0
32	34.7	112.0	20.7	42.9	48.4	250.3	3.3	54.9	1.1	9.8
64	17.1	56.7	10.4	24.0	44.5	148.0	1.6	50.4	1.3	7.1
128	8.5	29.1	5.9	15.6	30.8	123.0	1.4	38.5	1.6	N/A

Source Lines of Code (SLOC)



Experimental Results

C++ - Locality

- Intel Pentium 4, 3.0 GHz, 8KB L1

Matrix Size	Naïve 3 loops	Tiled 6 loops	HTA	HTA+ATLAS	ATLAS	Intel MKL(1)
504	161	657	675	2069	2387	3624
1008	150	649	679	2192	2384	3762
2016	133	632	675	2216	2492	3821
3024	135	644	668	2245	2509	3716
4032	36	588	613	2217	2519	3752

Scalability Challenges

- MATLAB
  - **Interpretation overhead**
  - **Vectorization**
  - **Copy-by-value**
  - **Poor data abstraction (e.g. LU and BT)**
- C++
  - **Runtime costs (e.g. virtual function calls)**
  - **Generic, Extensible design**
  - **Syntax**
- Language Independent
  - **Elimination of temporaries (e.g. stencil computations)**
  - **Scalable communication operations**

Related Works

Parallel PLs only

- A Spectrum of Parallel Programming Paradigms
  - **HPF, ZPL, CAF, UPC, POOMA, POET, CHAPEL**
- Various Parallel MATLAB implementations
  - **MATLAB*P, pMATLAB, MultiMATLAB etc**
  - **(This is out of scope)**
- Classification
  - **Languages versus Libraries**
  - **Global View of Data versus Segmented View**
  - **Single Thread of execution (ST) versus Multiple Threads (MT)**

Comparison

Approach	Implementation		Address Space	Control
	Language	Library		
CAF	√		Global	MT
GAS		√	Global	MT
HPF	√		Global	ST
HTA		√	Global	ST
MPI/PVM		√	Local	MT
POET		√	Global	MT
POOMA		√	Global	ST
Titanium	√		Global	MT
UPC	√		Global	MT
X10	√		Global	MT
ZPL	√		Global	MT

Conclusion

- HTA as a new data abstraction for modern systems
  - **F90 like high-level operations**
  - **Explicitness for performance control**
- Unification of
  - **Parallelism + Locality**
  - **Program Portability + Performance Portability**
  - **Sequential + Parallel Programming**
- Extension of existing languages
  - **C++ and MATLAB**
- Future Work
  - **Porting NAS benchmarks to C++ (Under Progress)**
  - **Scalability tests**
  - **Stronger cases for Hierarchical Tiling**

Thank You

```
void  conj_grad (SparseMatrix A, Matrix x, Matrix z, double& rnorm) {
  z = 0.0;  q = 0.0;        /* Conjugate Gradient Benchmark in CG */
  r = x;  p = r;
  rho = Matrix::dotproduct(r, r);
  for(int i = 0; i < CGITMAX; i++) {
    qt = 0;
    pt = p.ltranspose();
    SparseMatrix::lmatmul(A, pt, &qt);
    qt = qt.reduce(plus<double>(), 0.0, 1, 0x2);
    q = qt.transpose();
    alpha = rho / Matrix::dotproduct(p, q);
    z = z + alpha * p; r = r - alpha * q;
    rho0 = rho;
    rho = Matrix::dotproduct(r, r);
    beta = rho / rho0;
    p = r + beta * p;
  }
  qt = 0;   /* compute the norm*/
  Matrix zt = z.ltranspose();
  SparseMatrix::lmatmul(A, zt, &qt);
  qt = qt.reduce(plus<double>(), 0.0, 1, 0x2);
  r = qt.transpose();
  Matrix f = x - r;
  f = f * f;
  norm = f.reduce(plus<double>(), 0.0);
  rnorm = sqrt(norm);
}
```