Generalized Multipartitioning for Multi-Dimensional Arrays

Daniel Chavarría, Alain Darte (CNRS, ENS-Lyon),
Robert Fowler, John Mellor-Crummey

Department of Computer Science
Rice University

IPDPS 2002

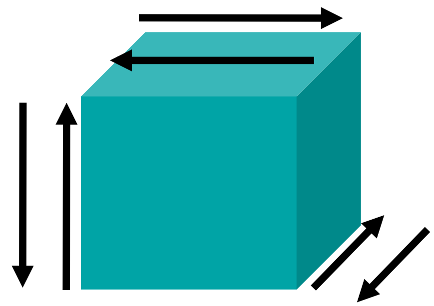
Supported by DOE/LACSI

Outline

- Line-sweep computations
  - multipartitioning, a sophisticated data distribution that enables better parallelization
- Generalized multipartitioning
  - objective function
  - find partitioning (number of cuts in each dimension)
  - map tiles to processors
- Performance results using the dHPF compiler
- Summary

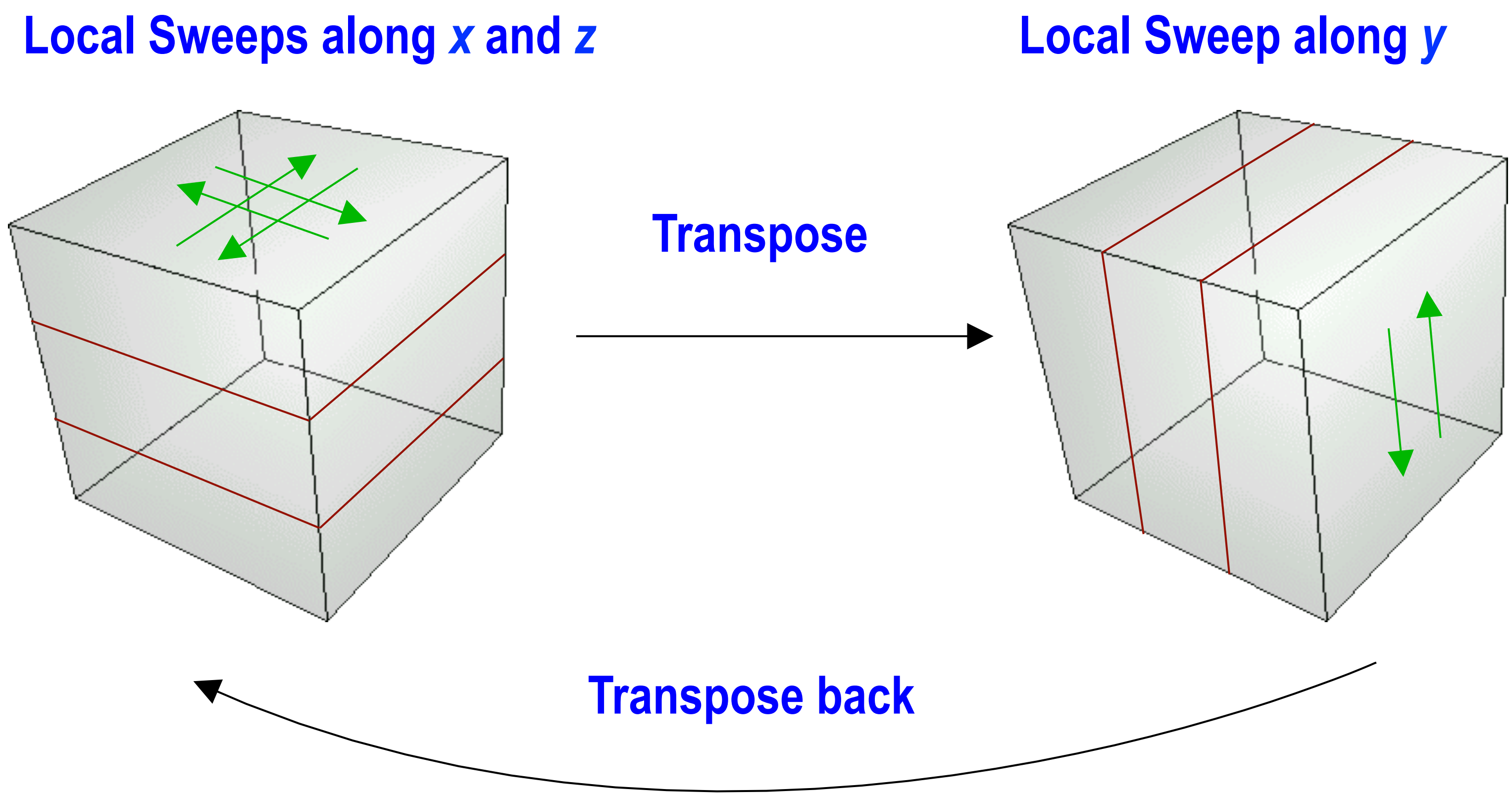
Line Sweep Computations

- 1D recurrences on a multidimensional domain



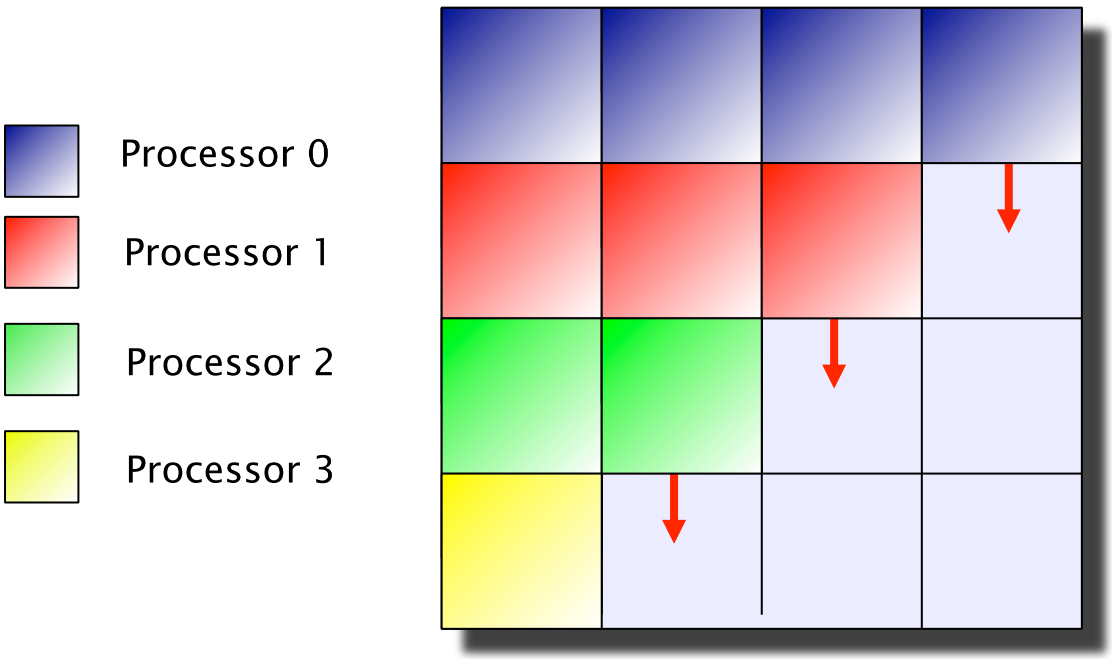
- Recurrences order computation along each dimension
- Compiler based parallelization is hard: loop carried dependences, fine-grained parallelism

Partitioning Choices: Transpose



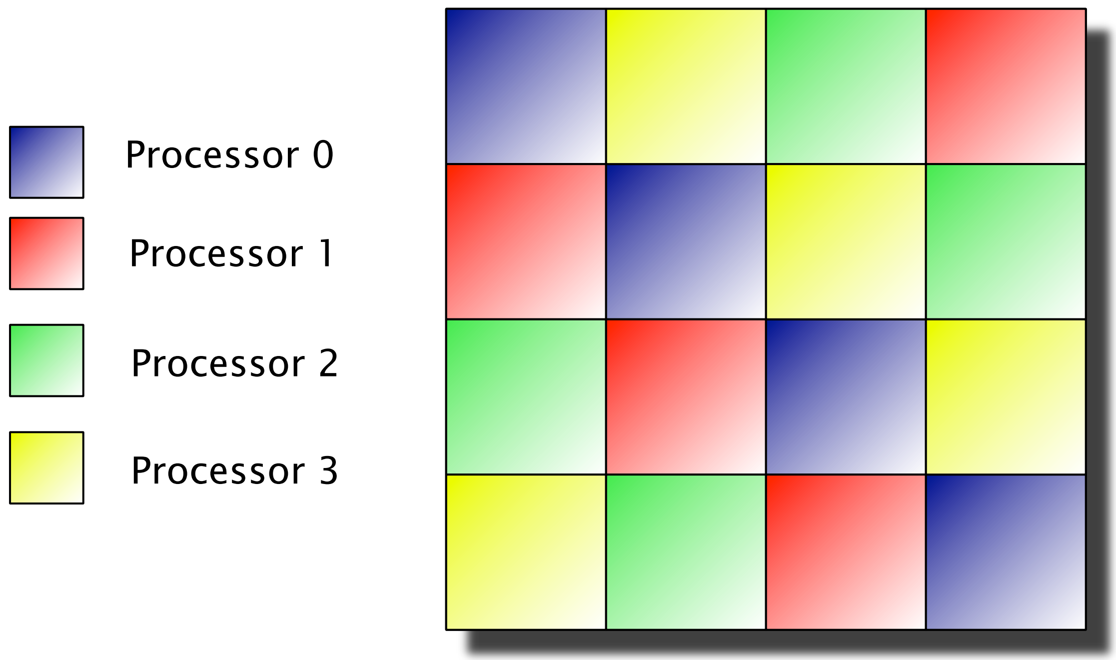
Partitioning Choices: Block + CGP

- Partial wavefront-type parallelism



Multipartitioning

- Style of skewed-cyclic distribution
- Each processor owns a tile between each pair of cuts along each distributed dimension

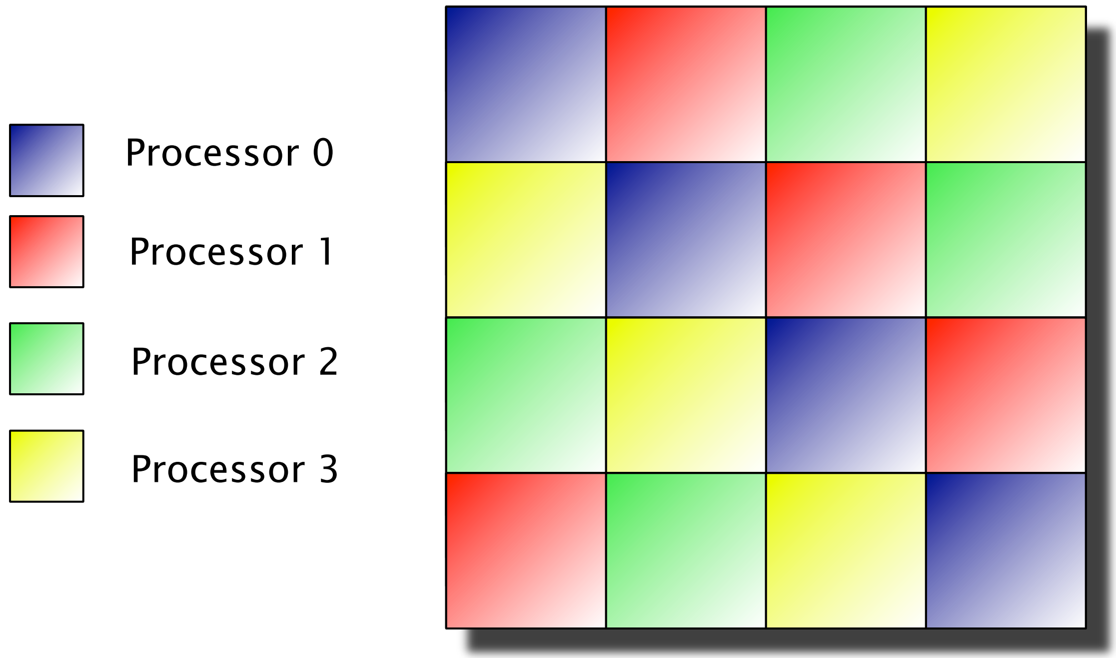


Multipartitioning

+ Full parallelism for a line-sweep computations

+ Coarse-grain communication

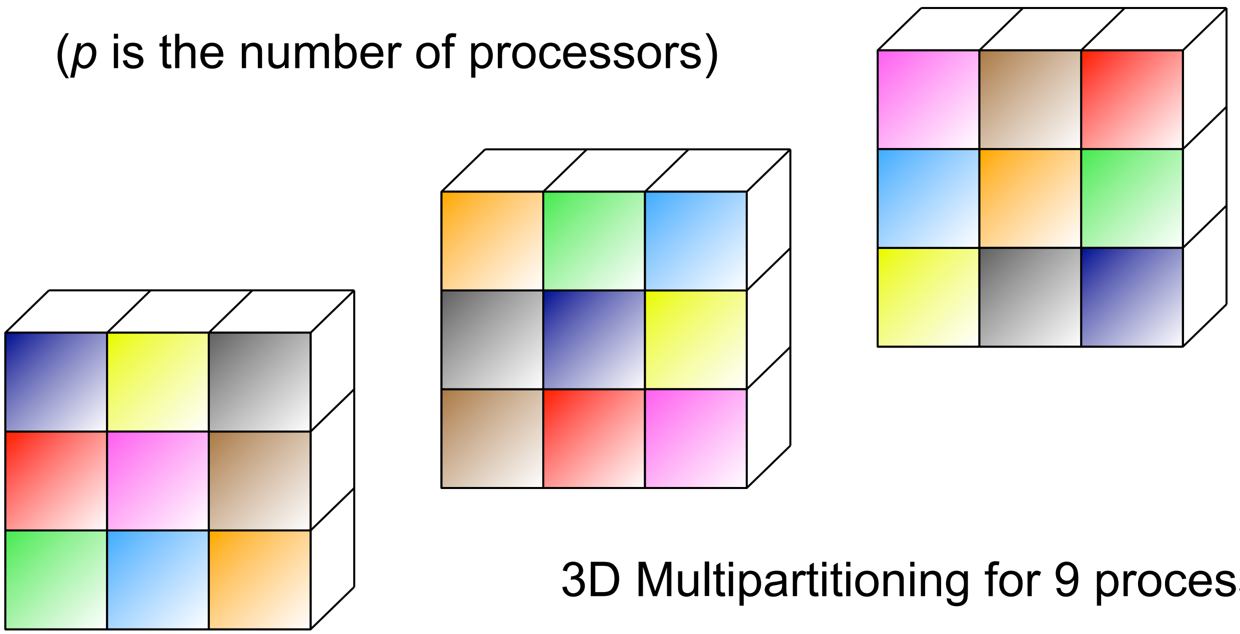
– Difficult to code by hand



Higher-dimensional Multipartitioning

- An array of k > d dimensions can be partitioned into $p^{d/(d-1)}$ tiles (diagonal multipartitioning)

(*p* is the number of processors)



3D Multipartitioning for 9 processors

Multipartitioning: Restrictions

In 3D, array of blocks of size b_1, b_2, b_3

One tile per processor per slice ➔ $p = b_1 b_2 = b_2 b_3 = b_1 b_3$

Thus: $\boxed{b_1 = b_2 = b_3}$

➔ the number of processors is a square, and the number of cuts in each dimension is $\sqrt{p}$.

In 3D, (standard) multipartitioning is possible for 1, 4, 9, 16, 25, 36, … processors.

What if we have 32 processors? Must we use only 25?

Outline

- Line-sweep computations
  - multipartitioning, a sophisticated data distribution that enables better parallelization
- ☞ Generalized multipartitioning
  - objective function
  - find partitioning (number of cuts in each dimension)
  - map tiles to processors
- Performance results using the dHPF compiler
- Summary

Generalized Multipartitioning: More Tiles for each Processor

Given a data domain n_1 x … x n_d and p processors

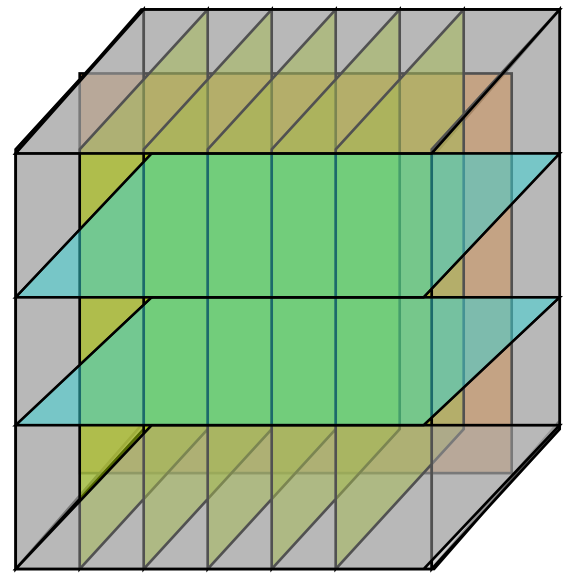
- Cut the array into b_1 x … x b_d so that, for each slice, the number of tiles is a multiple of the number of processors, i.e., for any i in [1..d], $\prod_{j \neq i} b_j$ is a multiple of p.
- Among valid partitionings, choose one that induces minimal communication overhead.
- Find a way to map tiles to processors so that:
  - in each slice, the same number of tiles is assigned to each processor (load-balancing property).
  - in any direction, the neighbor of a given processor is the same (neighbor property).

Objective Function for Multipartitioning

- Communication time depends upon the partitioning.

  Ex: p=6, array of 2x6x3 tiles.

$$\sum_{i=1}^{d} b_i k_i$$

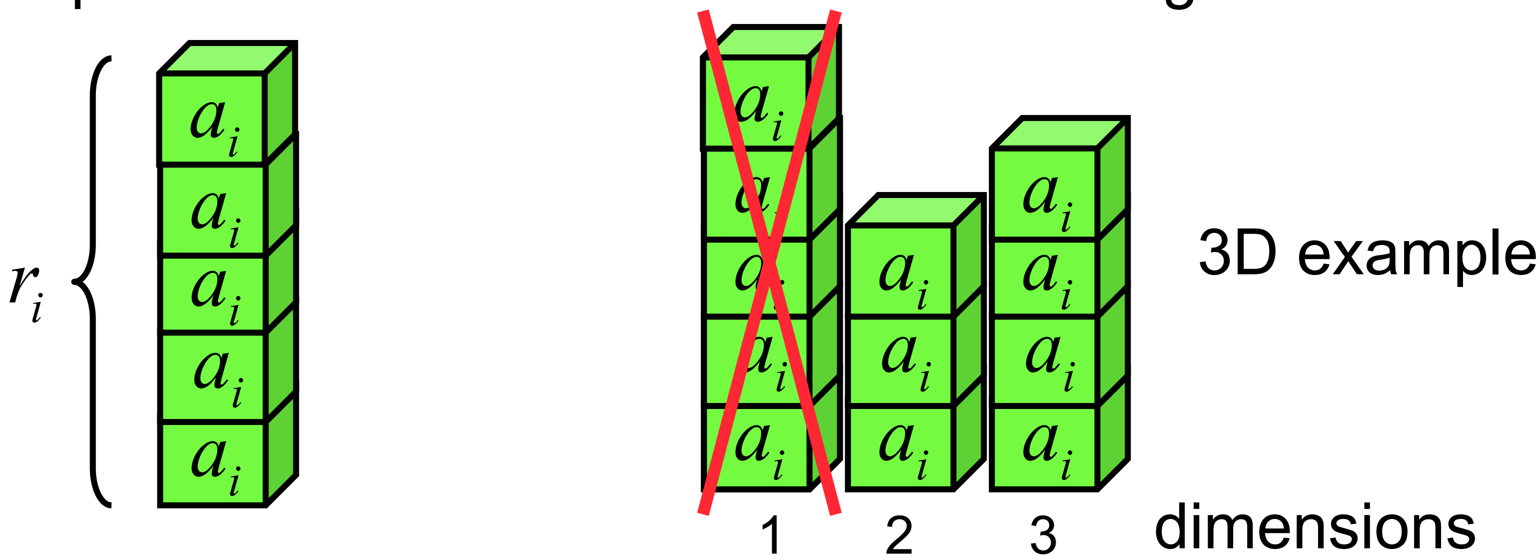


- Communication phases should be minimized.

➔The b_i's (number of cuts) should be large enough so that there are enough tiles per slice, but should be minimized to reduce the number of communication phases.

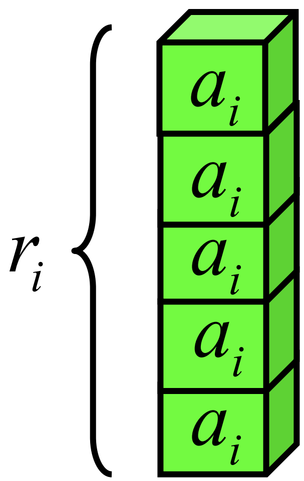
Elementary Multipartitioning

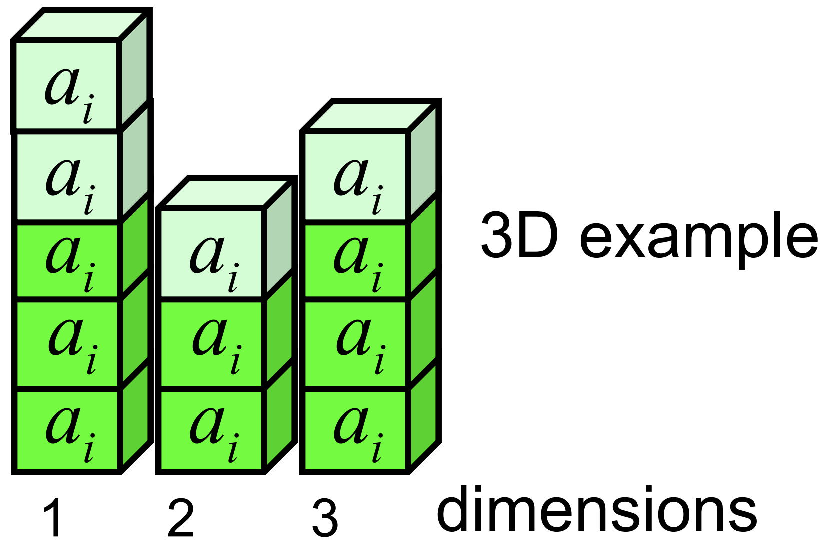
- Identify all sizes that are not "multiple" of smaller sizes. Decompose p into prime factors $p = (a_1)^{r_1} \ldots (a_s)^{r_s}$ and interpret each dimension as a bin containing such factors.



- Property 1: valid multipartitioning iff each prime factor with multiplicity r appears at least r+m times in the bins, where m is the maximal number of its occurrences in any bin.

Elementary Multipartitioning

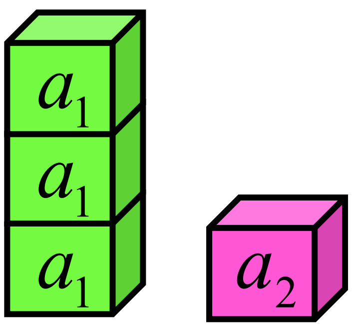


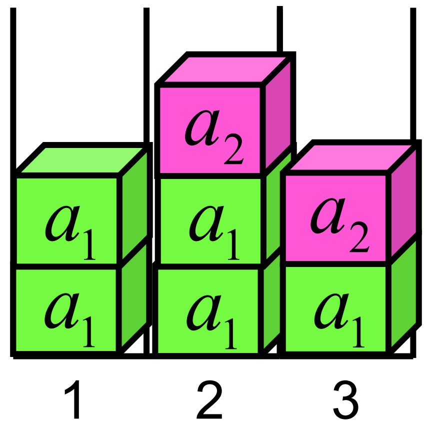


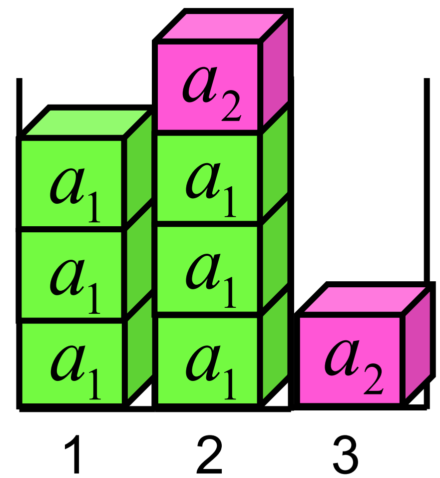
- If only one maximum, remove one in this maximal bin
  ➔ (r+m)-1 = r+(m-1) copies, thus a valid solution.
- If #elements > r+m, remove one anywhere ➔ still valid.
- Property 2: in an elementary multipartitioning, the total number of occurrences is exactly r+m, and the maximum m is reached for at least two bins.

Multipartitioning Choices

- Several elementary solutions:
  - For each factor: r+m=2m+e with 0 ≤ e ≤ (d-2)m ➔ r/(d-1) ≤m ≤ r.
  - Combine all factors.
- Example: $p = a_1^{\;3} a_2^{\;1}$







... plus all permutations

p = 8x3=24 ➔ solutions: 4x12x6, 8x24x3, 12x12x2, 24x24x1, ...

Algorithm for One Factor

(Algorithm similar to the generation of all partitions of an integer, see Euler, Ramanujam, Knuth, etc.).

```
Partitions(int r, int d) {
  for (int m= ⌈r/(d-1)⌉; m<=r; m++)     /* choose the maximal value */
    P(r+m,m,2,d);
}

P(int n, int m, int c, int d) { /* n elements in d bins, max m for at least c bins */
  if (d==1) bin[0]=n;           /* no choice for the first bin */
  else {
    for (int i=max(0,n-m(d-1)); i<=min(m-1,n-cm); i++) {
      bin[d-1]=i; P(n-i,m,c,d-1);  /* not maximum in bin number d-1 */
    }
    if (n>=m) {
      bin[d-1]=m; P(n-m,m,max(0,c-1),d-1); /* maximum in bin number d-1 */
    }
  }
}
```

Complexity of Exhaustive Search

- Naïve approach:

  For each dimension, choose a number between *1* and *p*, check that the load-balancing property holds, compute the sum, pick the best one.

  Complexity: more than p^d.

- By enumeration of elementary solutions:

  Generate only the tuples that form an elementary solution, compute the sum, pick the best one.

  Complexity:

  $$\left(\frac{d(d-1)}{2}\right)^{\frac{(1+o(1))\log p}{\log\log p}}$$

  and very fast in practice.

Possible unique factors of P

Number of choices for picking a pair of dimensions to partition with a number of cuts divisible by a particular prime factor

Outline

- Line-sweep computations
  - multipartitioning, a sophisticated data distribution that enables better parallelization
- Generalized multipartitioning
  - objective function
  - find partitioning (number of cuts in each dimension)
  - ☞ map tiles to processors
- Performance results using the dHPF compiler
- Summary

Tile-to-processor Mapping

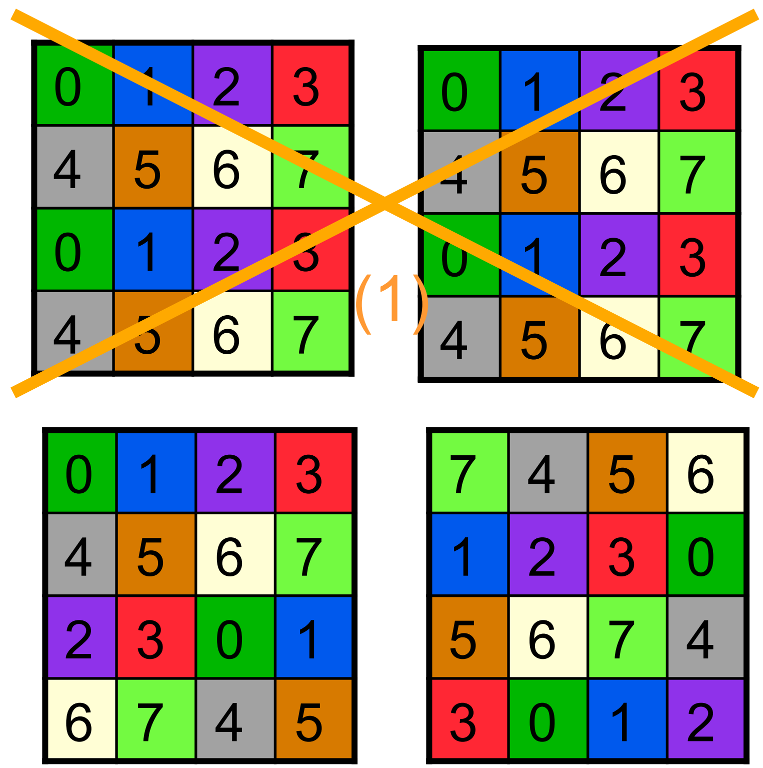
- So far, we have ensured that the number of tiles in each slice is a multiple of p. Is this sufficient? We want:

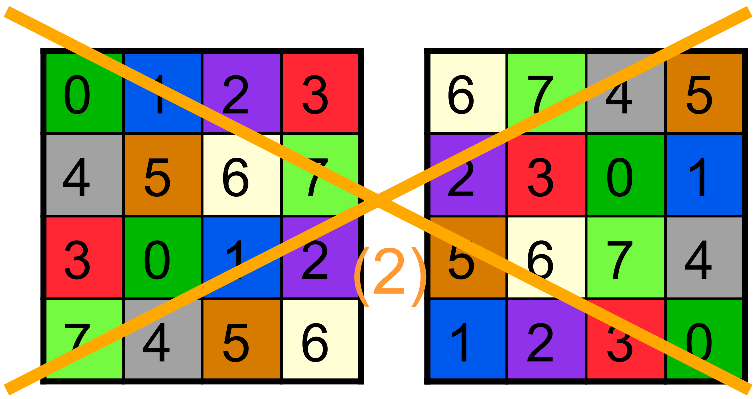
  (1) same number of tiles per processor per slice

  (2) neighbor property

- Example, 8 processors in 3D

  with 4x4x2 tiles.





Latin Squares and Orthogonality

In 2D, well-known concepts:

0	1	2	3
2	3	0	1
3	2	1	0
1	0	3	2

0	3	1	2
2	1	3	0
3	0	2	1
1	2	0	3

2 orthogonal diagonal latin squares.

When superimposed… magic squares!

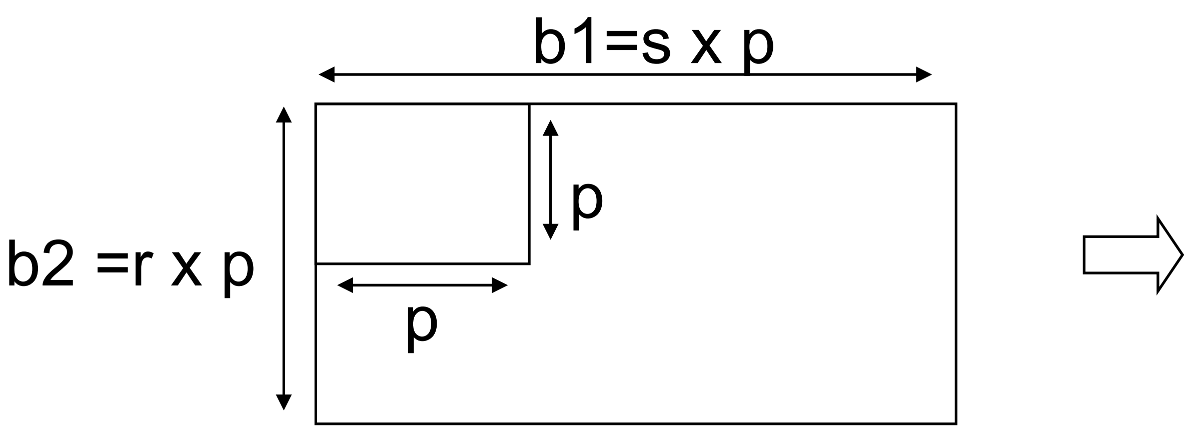
0,0	1,3	2,1	3,2
2,2	3,1	0,3	1,0
3,3	2,0	1,2	0,1
1,1	0,2	3,0	2,3

equivalent to (in base 4)

0	7	9	14
10	13	3	4
15	8	6	1
5	2	12	11

Dim ≥ 3 + Rectangles = Difficulties

- In any dimension, latin hypercubes are easy to build.
- In 2D, a latin rectangle can be built as a “multiple” of a latin square:



Pave with copies of latin squares.

- Not true in dim ≥ 3!

  Ex: for p=30, 10x15x6 is elementary, therefore it cannot be a multiple of any valid hypercube.

Mapping Tiles with Modular Mappings

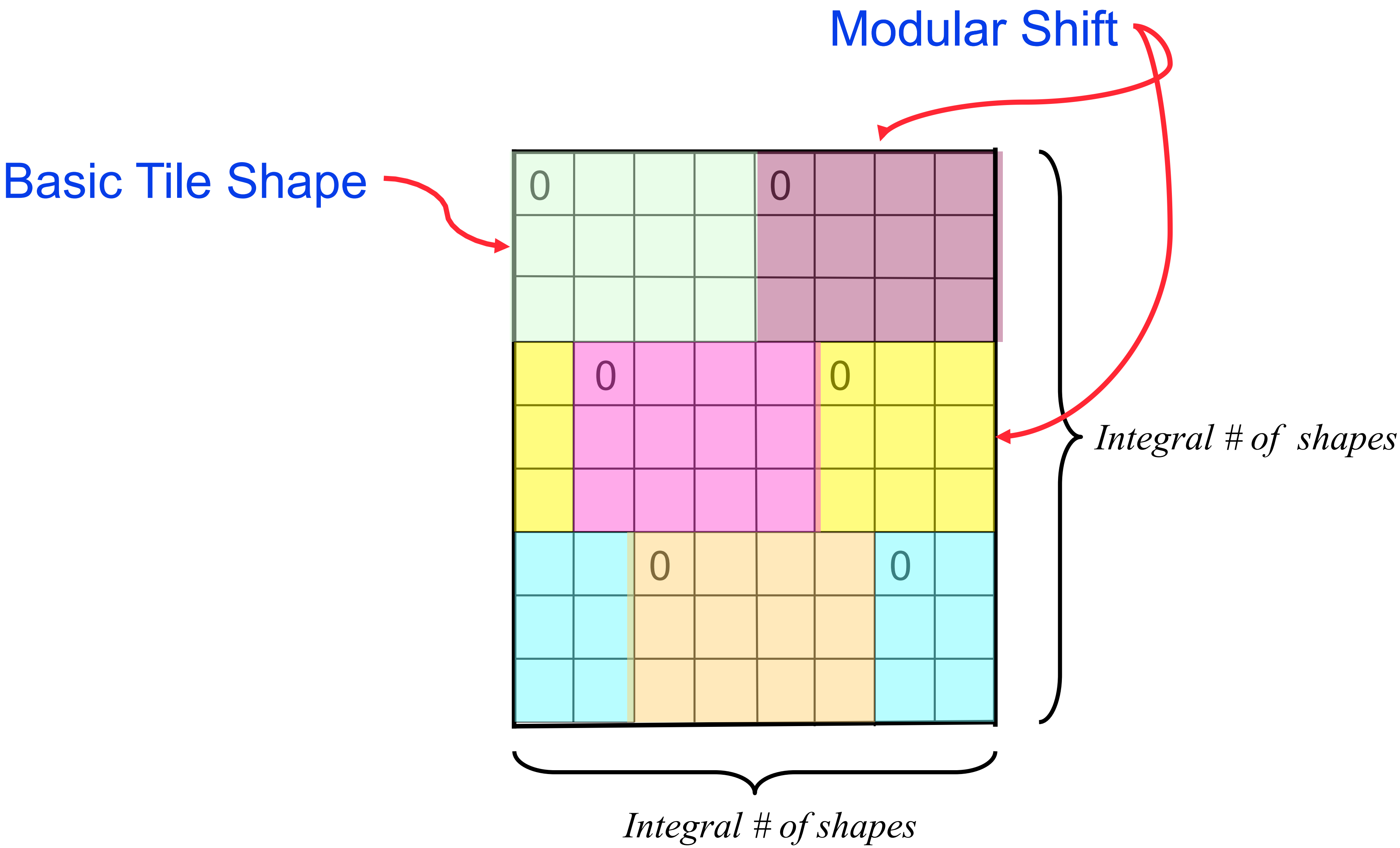
- Represent data tiles as a multi-dimensional grid
- Assign tiles with a linear mapping, modulo the grid sizes

Example: A modular mapping for a 3D multipartitioning

$$\text{Tile}\begin{pmatrix} i \\ j \\ k \end{pmatrix} \text{ on Processor } \begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \end{pmatrix}\begin{pmatrix} i \\ j \\ k \end{pmatrix} \bmod \begin{pmatrix} m_1 \\ m_2 \end{pmatrix}$$

(details in the paper)

A Plane of Tiles from a Modular Mapping



Modular Mapping: Solution

Long proofs but …
"simple" codes! ☺

Computation of m:

```
f=1; g=p;
for (i=1; i<=d; i++) {
  f = f*b[i];
}
for (i=1; i<=d; i++) {
  m[i]=g;
  f=f/b[i];
  g=gcd(g,f);
  m[i]=m[i]/g;
}
```

Computation of M:

```
for (i=1; i<=d; i++) {
  for (j=1; j<=d; j++) {
    if ((i==1) || (i==j)) M[i][j] = 1;
    else M[i][j]=0;
  }
}
for (i=1;i<=d;i++) {
  r=m[i];
  for (j=i-1; j>=2; j--) {
    t = r/gcd(r, b[j]);
    for (k=1; k<=i-1; k++)
      M[i][k] = M[i][k] - t*M[j][k];
    r = gcd(t*m[j],r);
  }
}
```

Outline

- Line-sweep computations
  - multipartitioning, a sophisticated data distribution that enables better parallelization
- Generalized multipartitioning
  - objective function
  - find partitioning (number of cuts in each dimension)
  - map tiles to processors
- Performance results using the dHPF compiler
- Summary

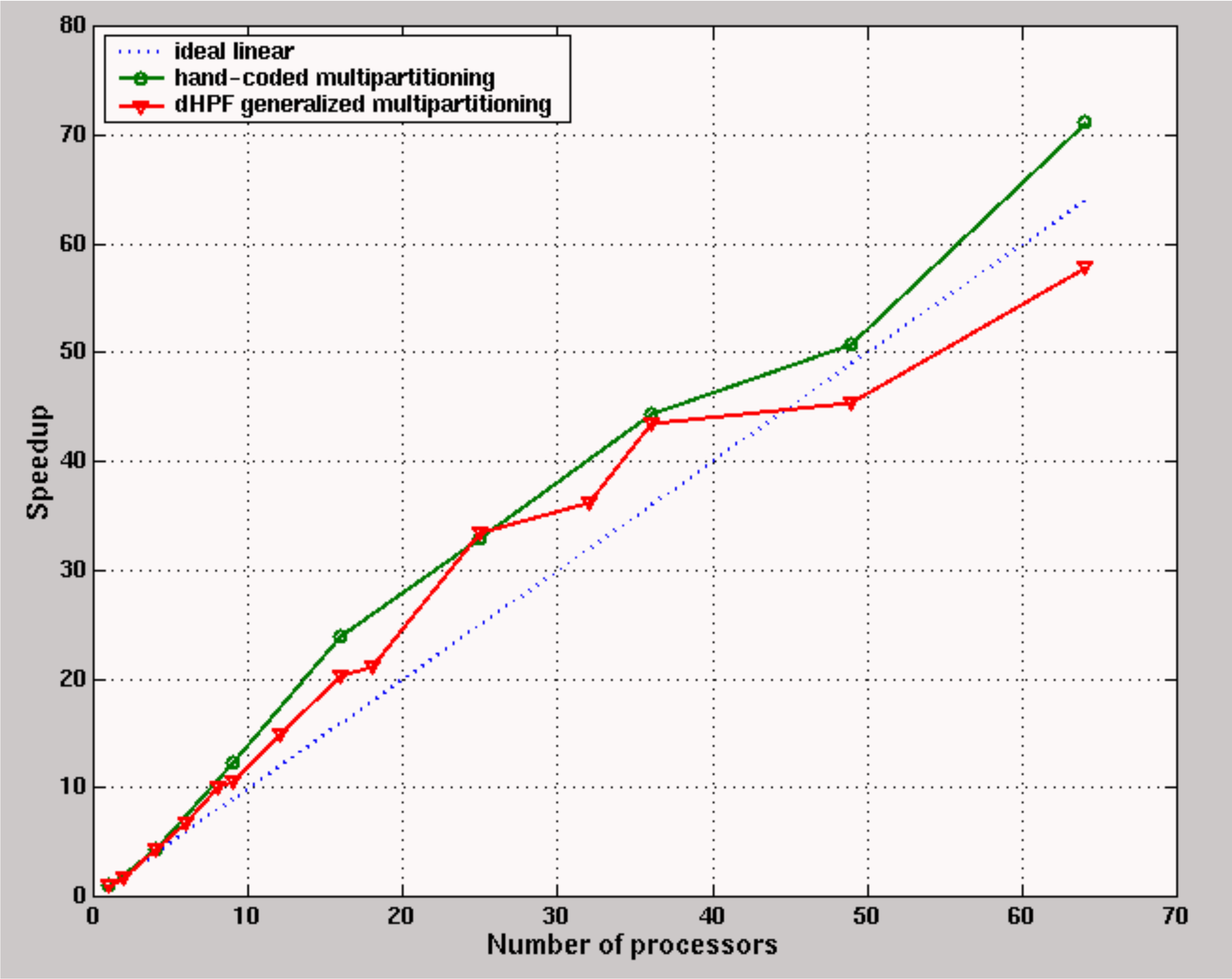
Compiler Support for Multipartitioning

- We have implemented automatic support for generalized multipartitioning in the dHPF compiler
  - Compiler takes High Performance FORTRAN (HPF) as input
    - » + MULTI data distribution directive
  - Outputs FORTRAN77 + MPI calls
- Support for generalized multipartitioning
  - Computes the optimal multipartitioning and tile mapping for p processors
  - Aggregates communication for multiple tiles & multiple arrays (by exploiting the neighbor property)

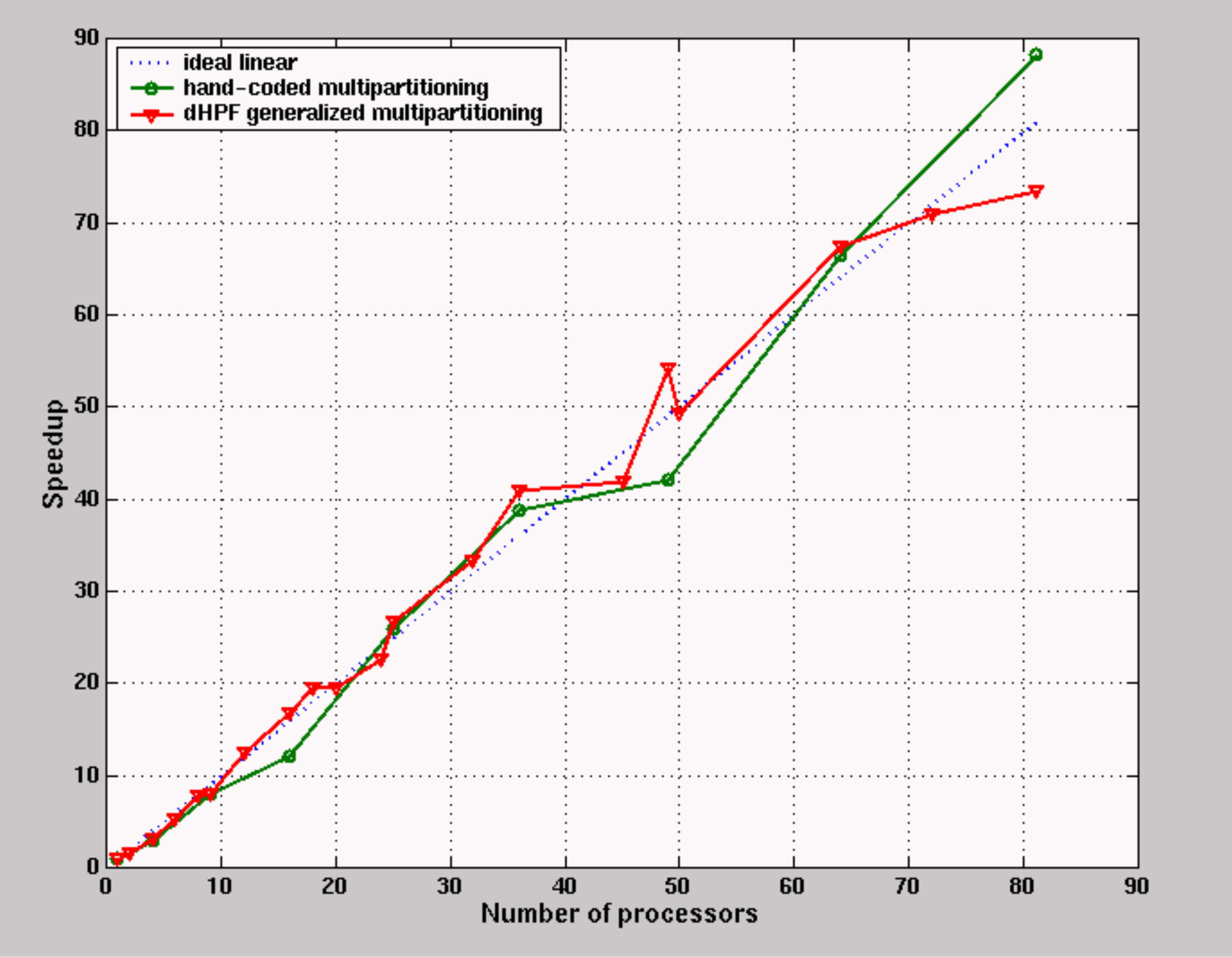
NAS SP Parallelization

- NAS SP benchmark
  - Uses ADI to solve the Navier-Stokes equation in 3D
  - Forward & backward line sweeps on each dimension, for each time step
- 2 versions from NASA, each written in FORTRAN77
  - Parallel MPI hand-coded version
  - Sequential version
- dHPF input: sequential version + HPF directives (including MULTI)
- Experimental platform: SGI Origin 2000, SGI's MPI implementation

NAS SP Speed-up (Class A)



NAS SP Speed-up (Class B)



Summary

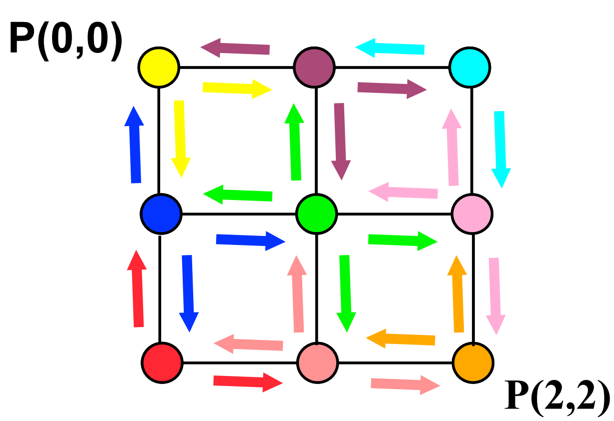
- A generalization of multipartitioning to any number of processors.
- A fast algorithm for selecting the best multipartitioning.
- A constructive proof for a suitable mapping (i.e., a "multi-dimensional latin hyper-rectangle") provided that the size of each slice is a multiple of p.
- New results on modular mappings.
- Complete implementation in the Rice dHPF compiler.
- Many open questions for mathematicians, related to the extension of Hajós theorem to "many-to-one" direct sums and "magic" squares, and to combinatorial designs.

HPF Program: Example

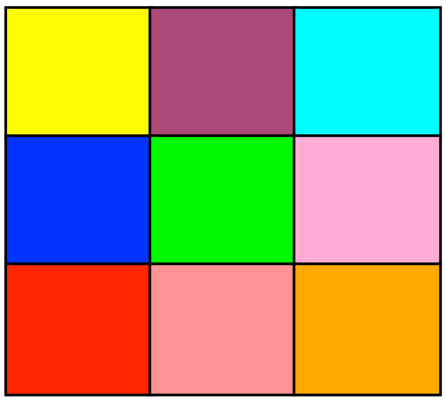
```
CHPF$ processors P(3,3)
CHPF$ distribute A(block, block) onto P
CHPF$ distribute B(block, block) onto P

    DO i = 2, n - 1
     DO j = 2, n - 1
        A(i,j) = .25 *(B(i-1,j) + B(i+1,j)+
                       B(i,j-1) + B(i,j+1))
```

High Performance Fortran

- Data-parallel programming style
- Implicit parallelism
- Communications generated by the compiler



Processors



Data for A, B

(BLOCK,BLOCK)distribution

dHPF Compiler at a Glance...

- Select computation partitionings
  - determine where to perform each statement instance
  - replicate computations to avoid communications
- Analyze and optimize communications
  - determine where communication is required
  - optimize communication placement
  - aggregate messages
- Generate SPMD code for partitioned computation
  - reduce loop bounds and insert guards
  - insert communication
  - transform references

Multipartitioning in the dHPF Compiler

- New directive for multipartitioning. Tiles are manipulated as virtual processors. Directly fits the mechanisms used for block distribution:
  - analyze messages with Omega Library.
  - vectorize both carried and independent communications.
  - aggregate communications.
    - » for multiple tiles (exploit "same neighbor" property)
    - » for multiple arrays
  - partial replication of computation to reduce communication.
- Carefully control code generation.
- Careful (painful) cache optimizations.

Objective Function

- One phase, several steps:

$$\text{computation} \longleftarrow \boxed{\tau_1 \frac{n}{p}} + (b_i - 1)\boxed{\left(\tau_2 + \tau_3 \frac{n}{p n_i}\right)} \longrightarrow \text{communication}$$

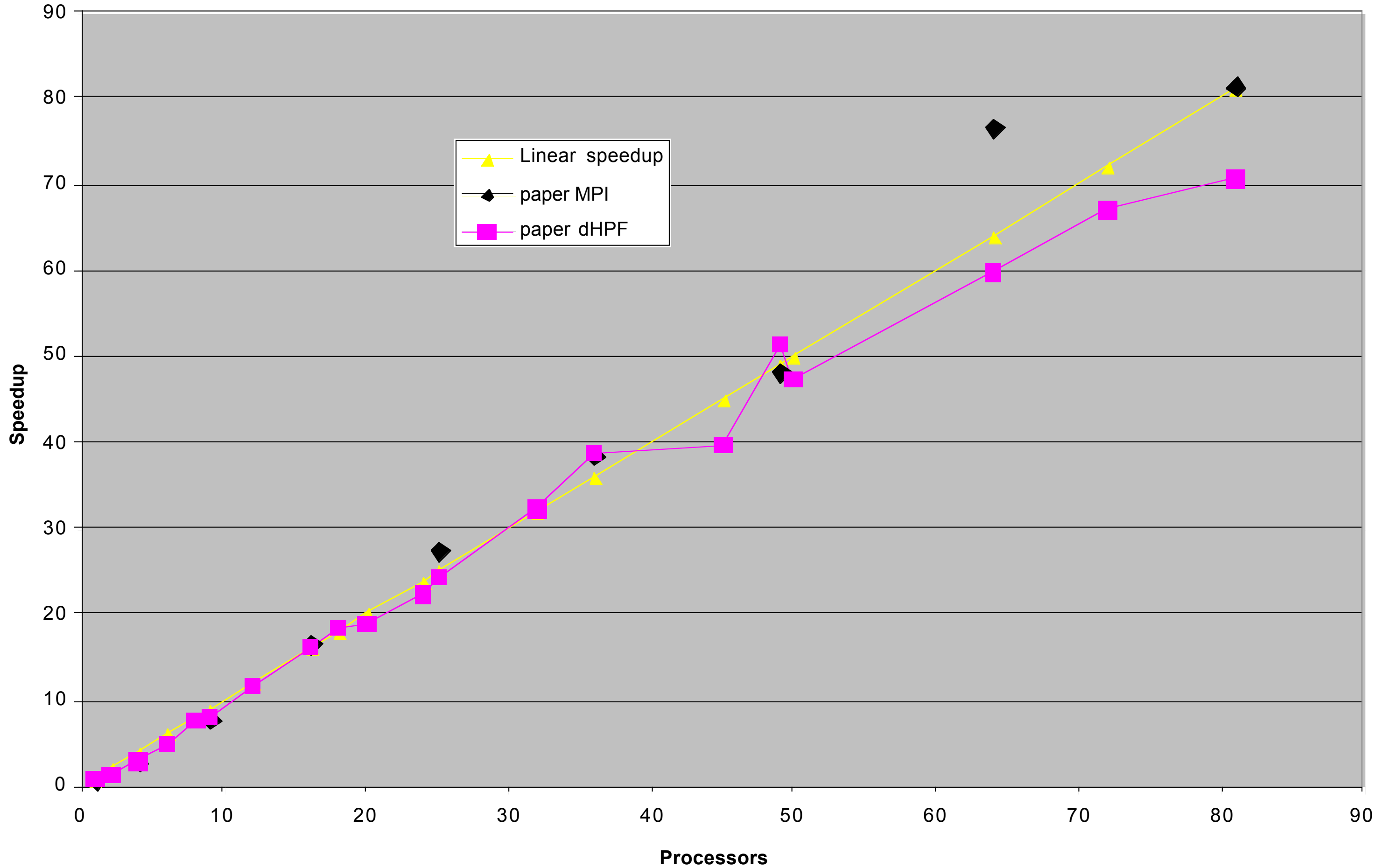
- All phases (one per dimension):

$$\underbrace{\sum_{i=1}^{d}\left(\left(\tau_1 \frac{n}{p}\right) - \left(\tau_2 + \tau_3 \frac{n}{p n_i}\right)\right)}_{\text{depends on p only}} + \underbrace{\sum_{i=1}^{d} b_i \left(\tau_2 + \tau_3 \frac{n}{p n_i}\right)}_{\text{objective function}}$$

- Try to minimize a linear function with positive parameters

$$\sum_{i=1}^{d} b_i k_i \quad \text{or} \quad \sum_{i=1}^{d} b_i$$

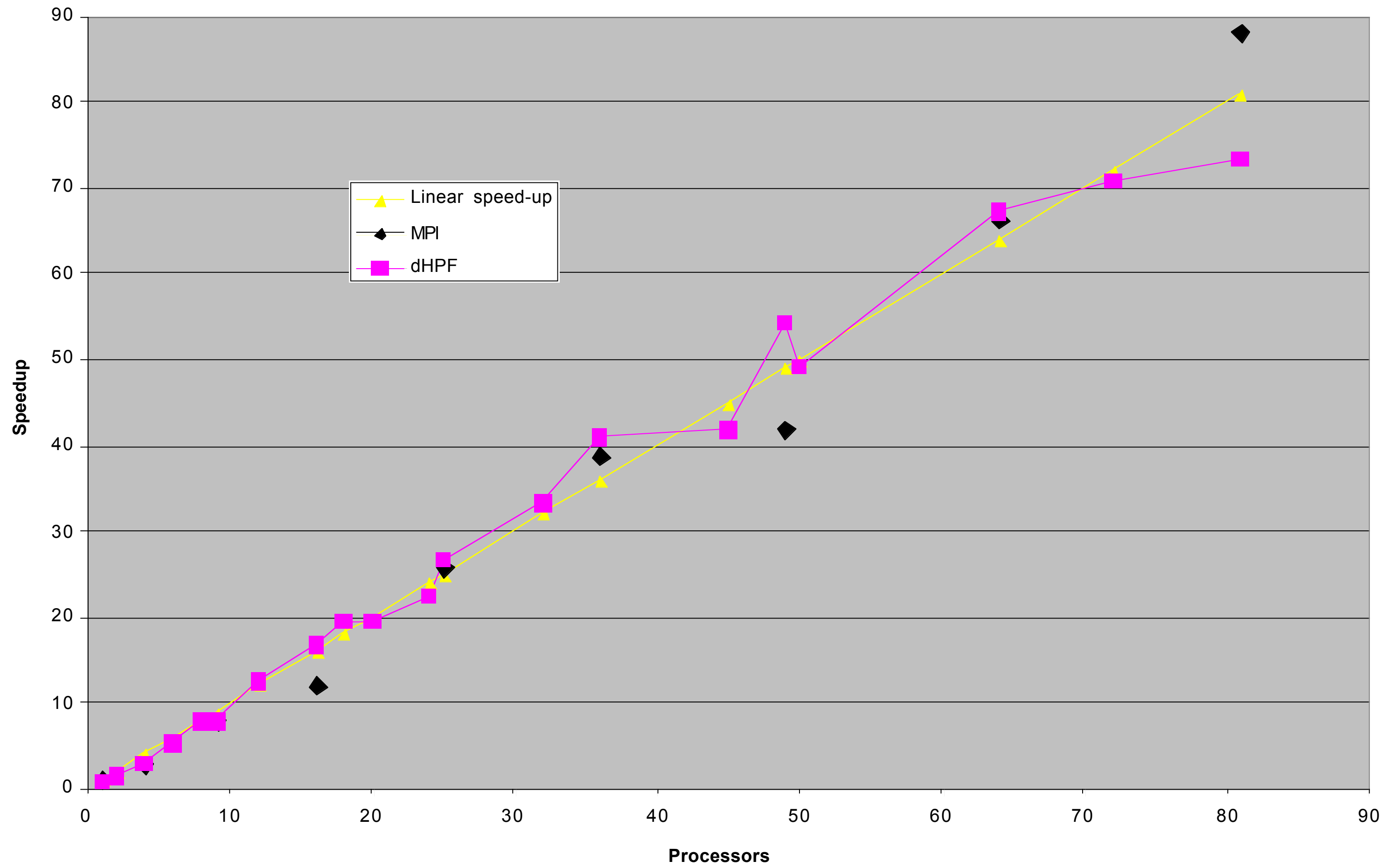
NAS SP Speed-up (Class B) using Generalized Multipartitioning



(June 2001)

SGI Origin 2000

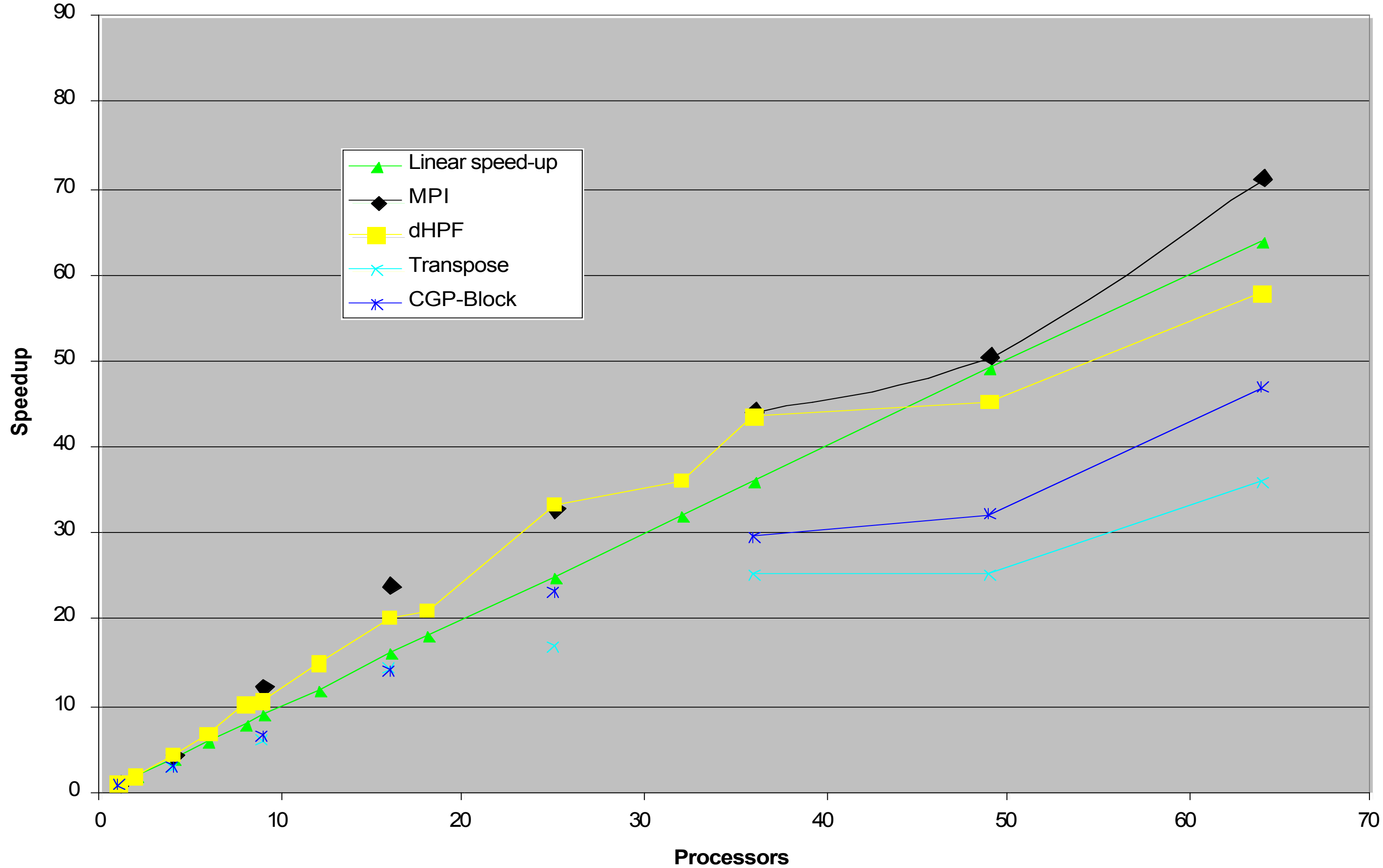
NAS SP Speedups (Class B) using Generalized Multipartitioning



(April 2002)

SGI Origin 2000

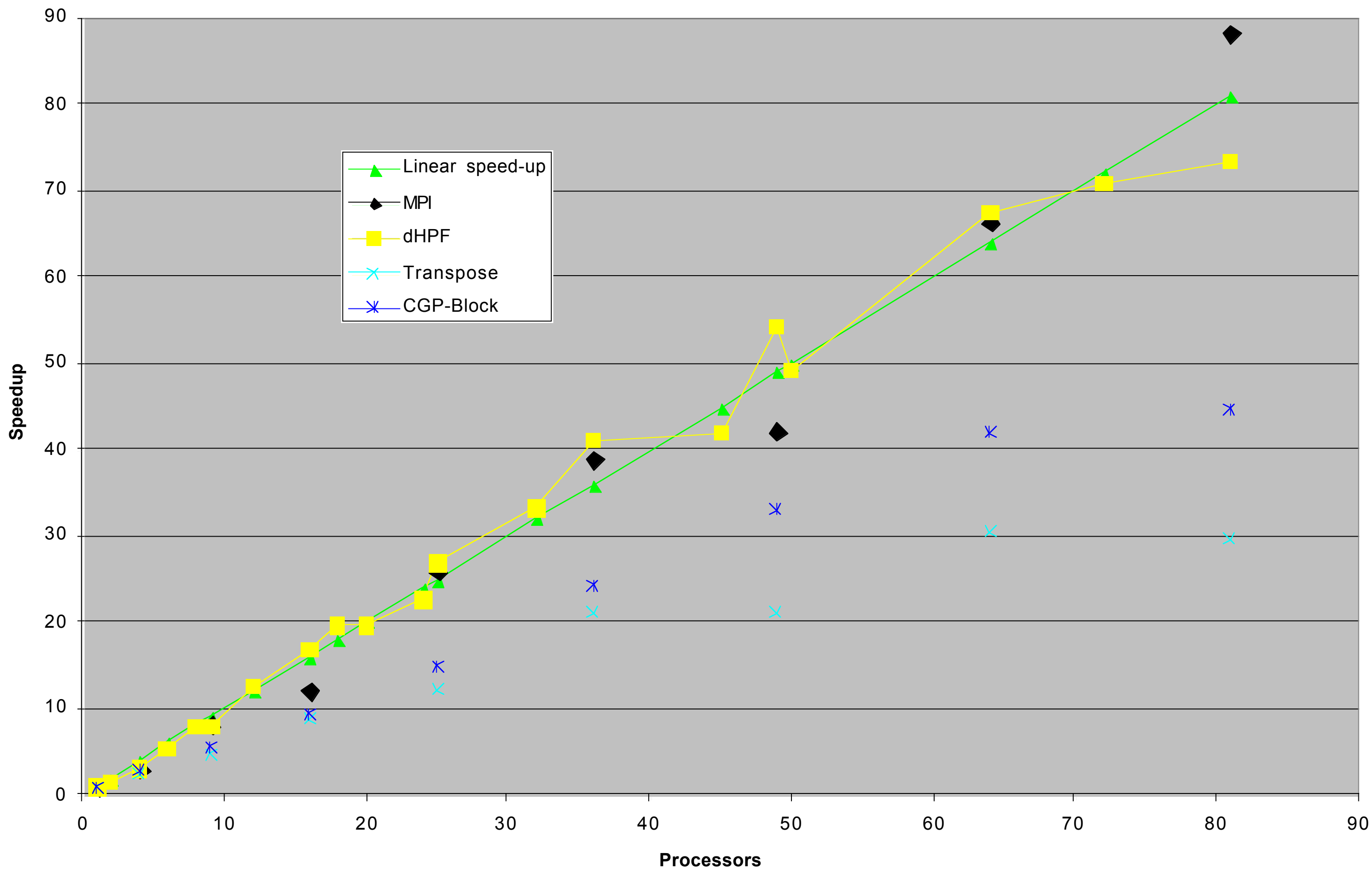
NAS SP Speedups (Class A) using Generalized Multipartitioning



(April 2002)

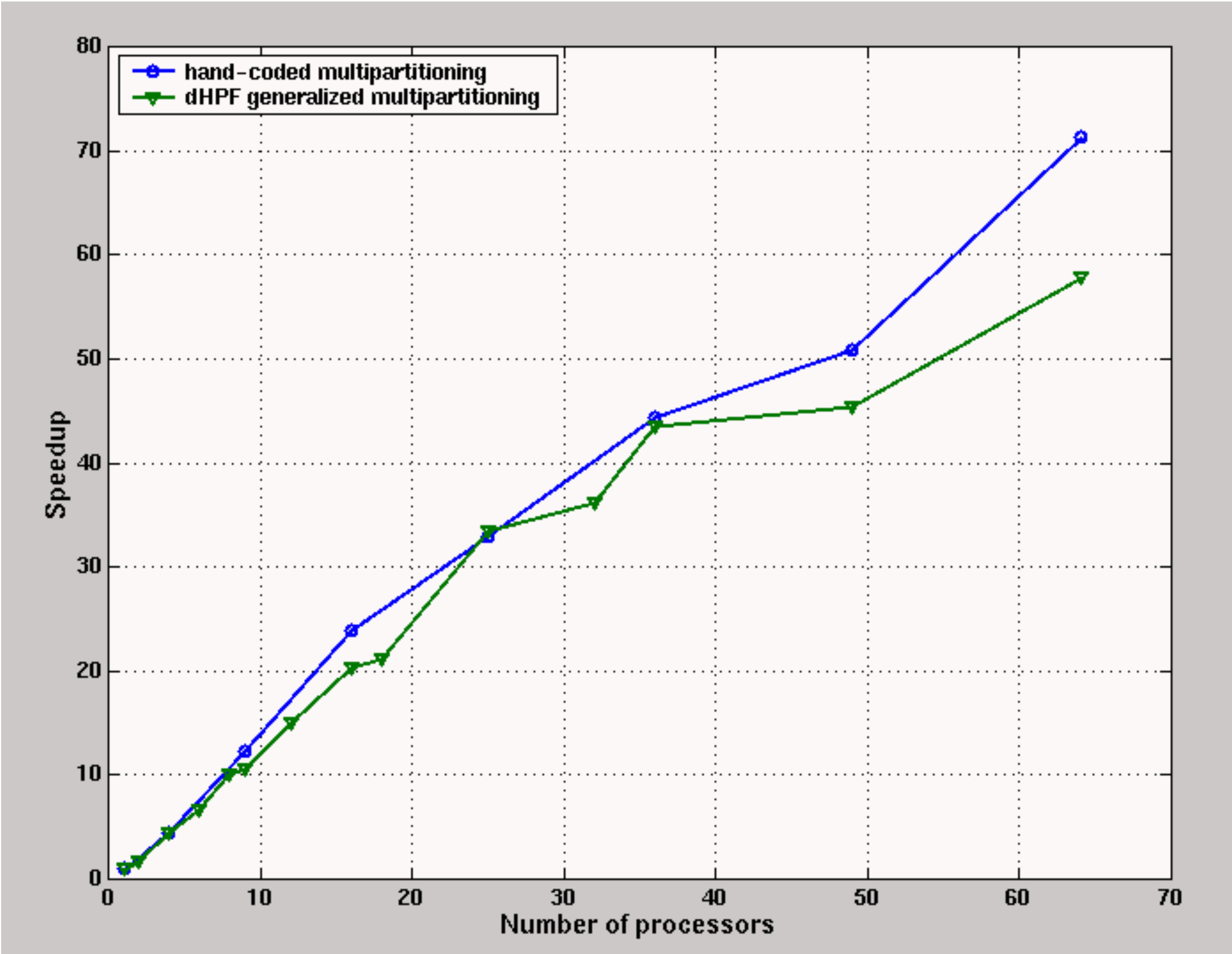
SGI Origin 2000

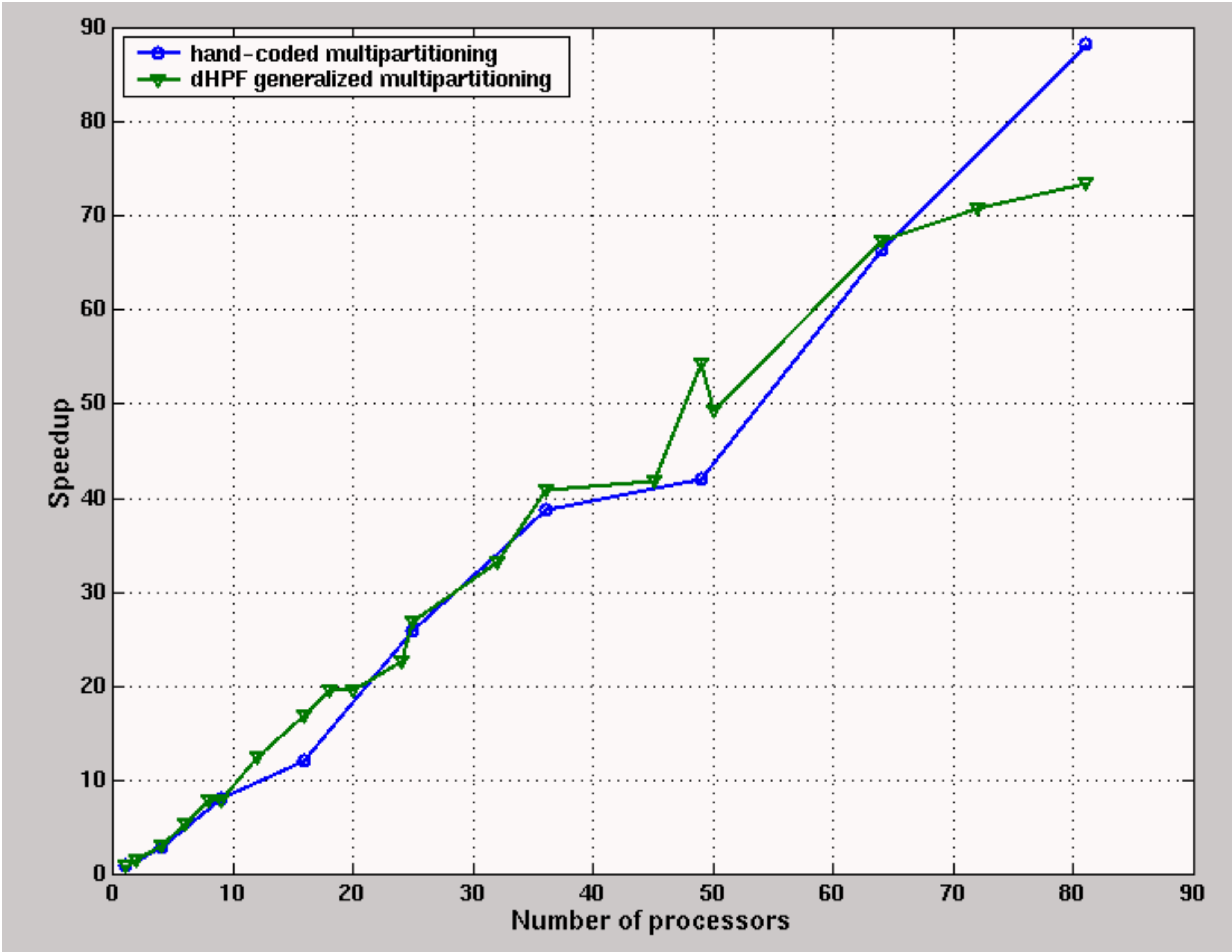
NAS SP Speedups (Class B) using Generalized Multipartitioning



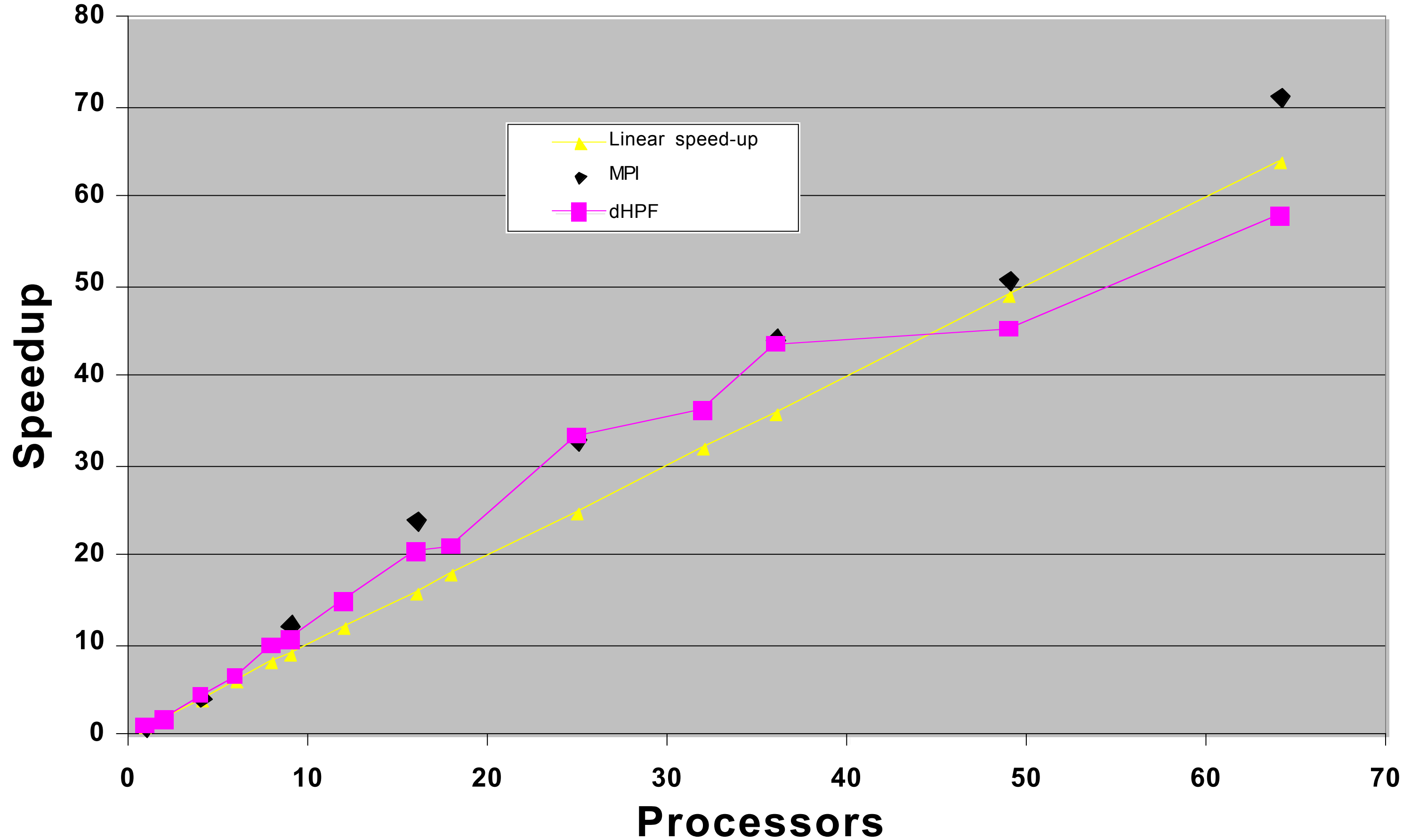
(April 2002)

SGI Origin 2000


hand-coded multipartitioning
dHPF generalized multipartitioning
Speedup
Number of processors
0
10
20
30
40
50
60
70
80


hand-coded multipartitioning
dHPF generalized multipartitioning
Speedup
Number of processors
0
10
20
30
40
50
60
70
80
90

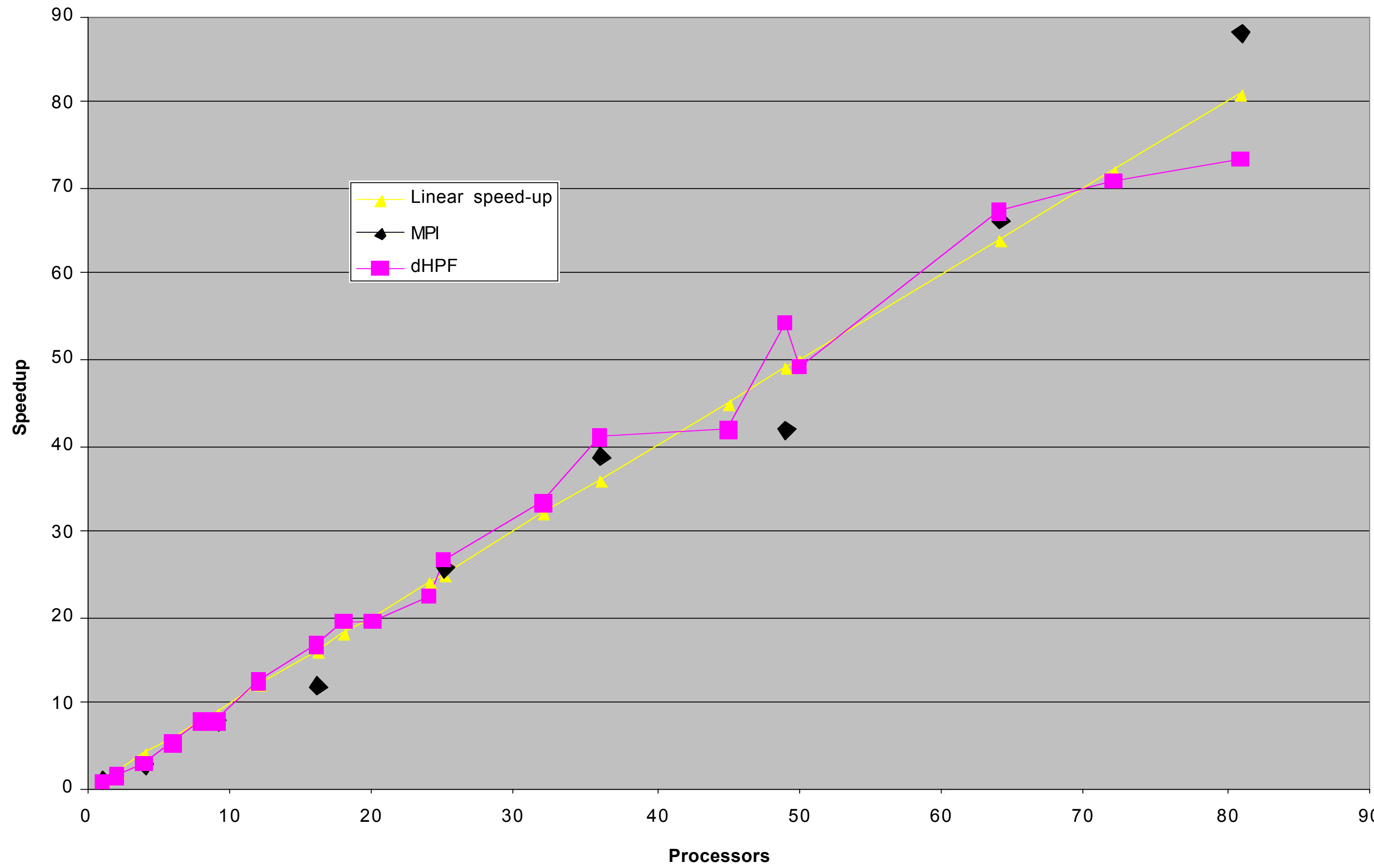
NAS SP Speed-up (Class A) using Generalized Multipartitioning



(April 2002)

SGI Origin 2000

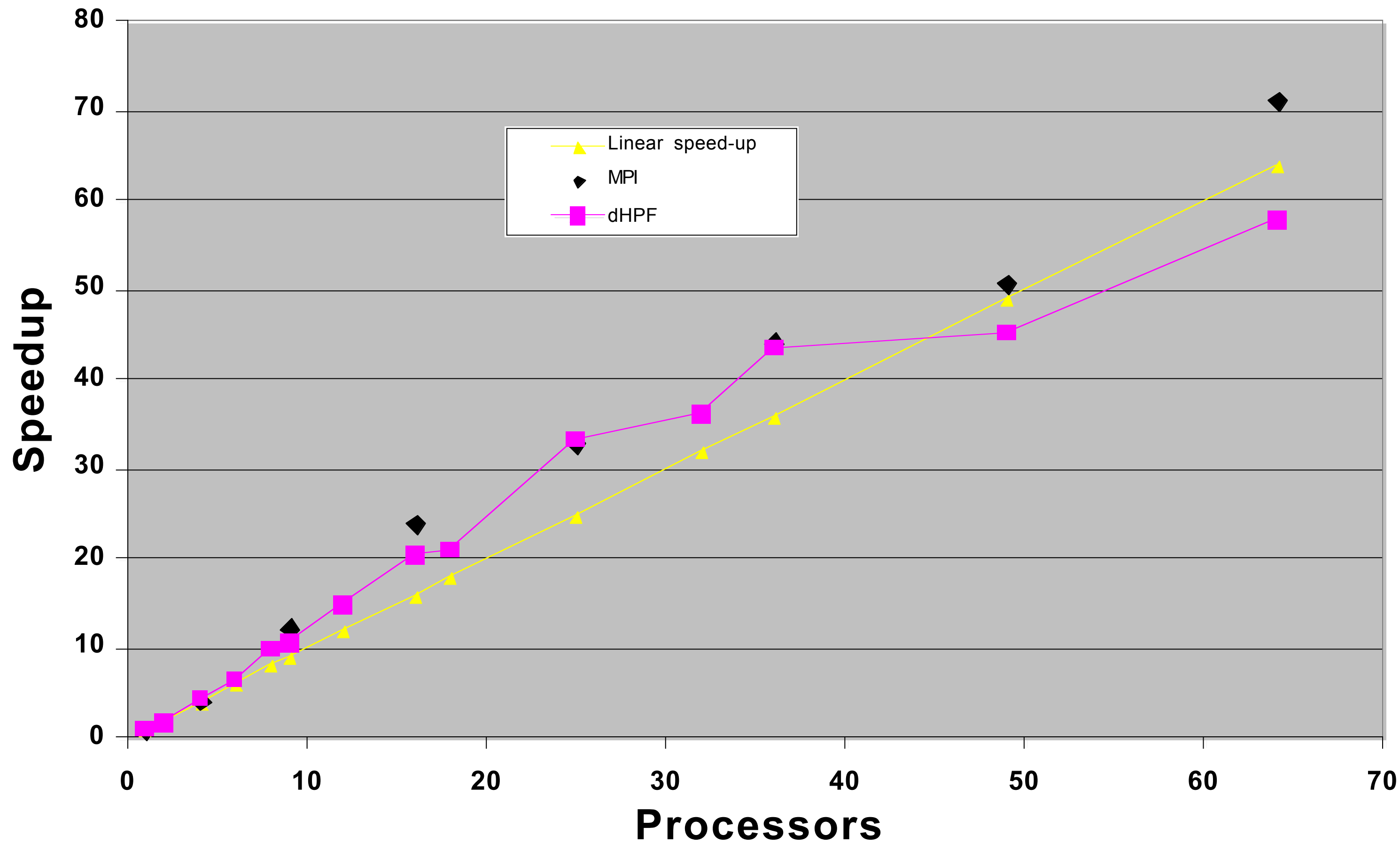
NAS SP Speedups (Class B) using Generalized Multipartitioning



(April 2002)

SGI Origin 2000

NAS SP Speed-up (Class A) using Generalized Multipartitioning

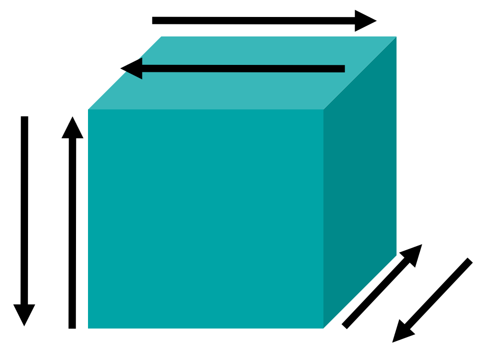


(April 2002)

SGI Origin 2000

Context

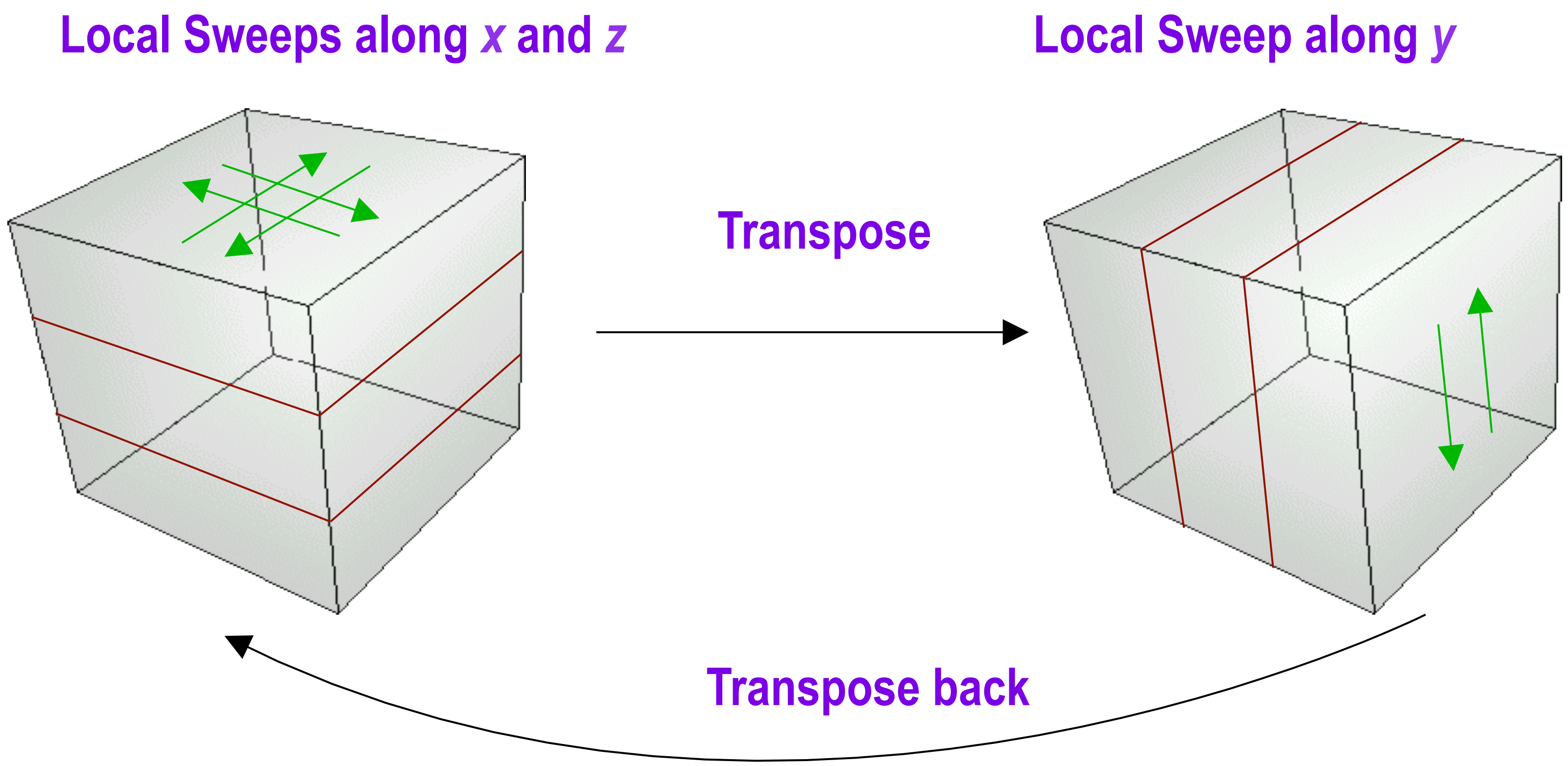
- Alternating Direction Implicit (ADI) Integration widely used for solving the Navier-Stokes equation in parallel and a variety of other computational methods [Naik'93].
- Structure of computation: line sweeps, i.e., 1D recurrences



- Compiler-based parallelization is hard: tightly coupled computations (dependences), fine-grain parallelism.
- Challenge: achieve hand-coded performance with dHPF (Rice HPF compiler).

Parallelizing Line Sweeps

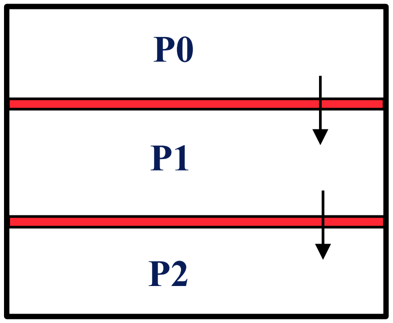
Approach 1: Avoid computing along partitioned dimensions



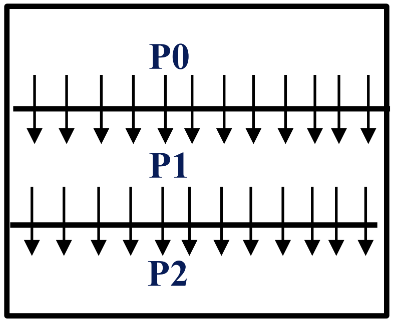
+ Fully parallel computation
– High communication volume: transpose ALL data

Parallelizing Line Sweeps

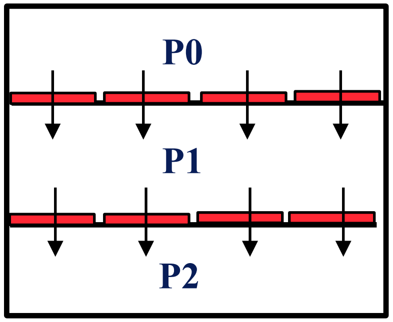
Approach 2: Compute along partitioned dimensions



- Loop carrying dependence in an outer position
  - ➔ Full serialization
  - ➔ Minimal communication overhead



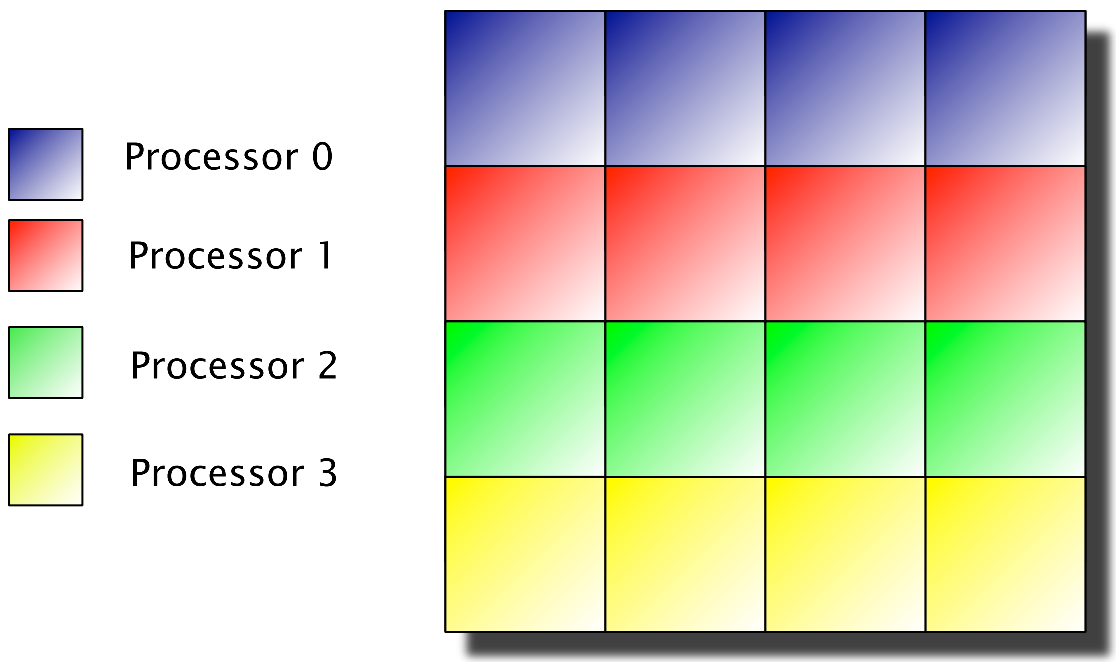
- Loop carrying dependence in innermost position
  - ➔ Fine-grained wavefront parallelism
  - ➔ High communication overhead



- Tiled loop nest, dependence at mid-level
  - ➔ Coarse-grained wavefront parallelism
  - ➔ Moderate communication overhead

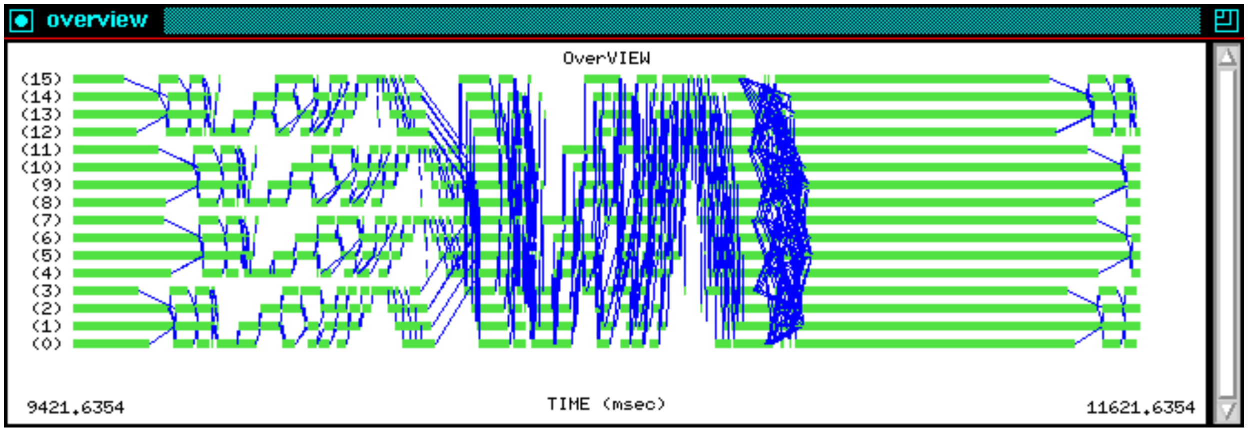
Coarse-grain Pipelining with Block Partitionings

- Wavefront parallelism
- Coarse-grain communication

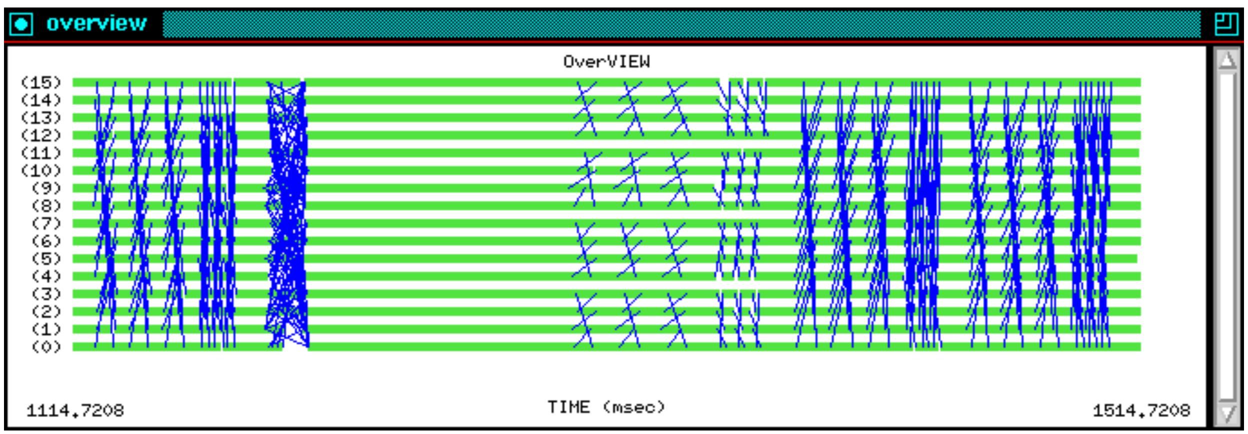


- Better performance than transpose-based parallelizations [Adve et al. SC98]

Parallelizing Line Sweeps



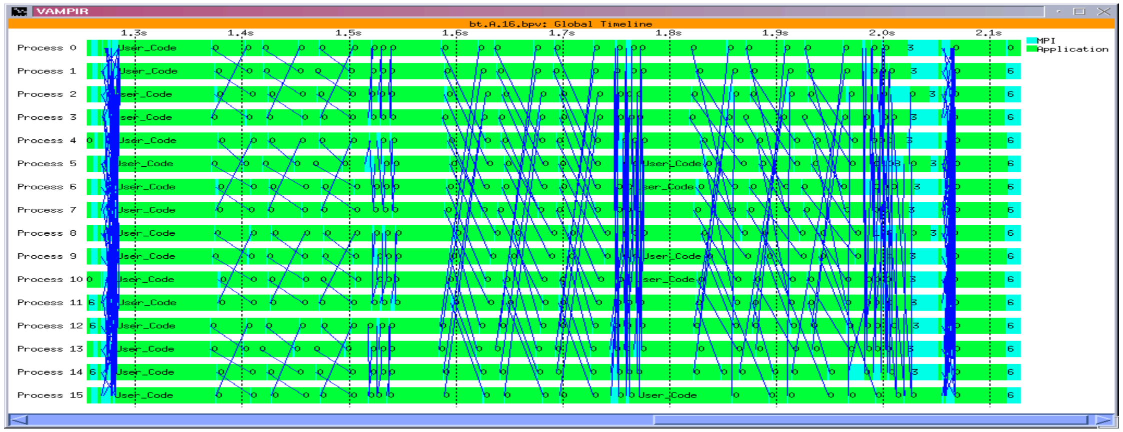
Compiler-generated coarse-grain pipelining



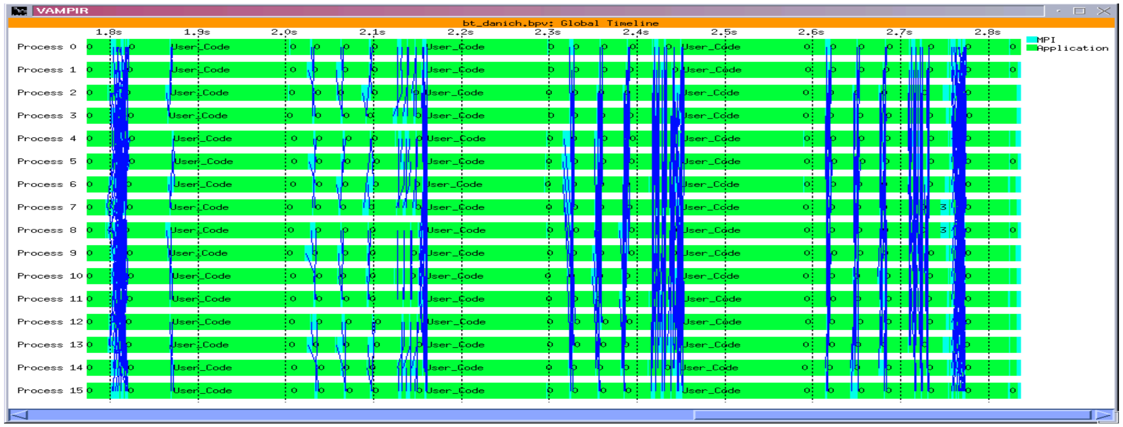
Hand-coded multipartitioning

NAS BT Parallelizations

Hand-coded
3D Multipartitioning



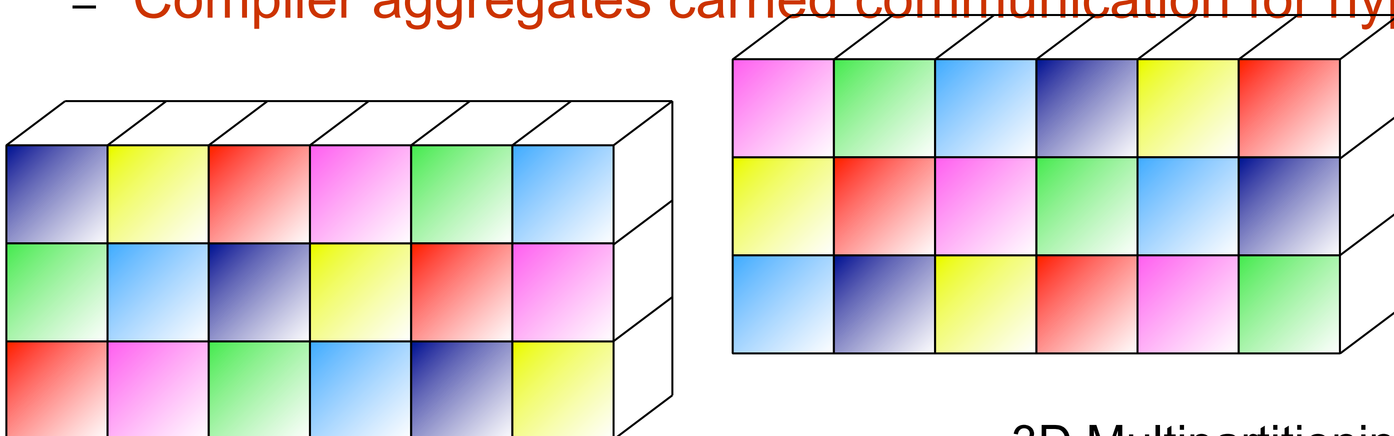
Compiler-generated
3D Multipartitioning
(dHPF, Jan. 2001)



Execution Traces for NAS BT Class 'A' - 16 processors, SGI Origin 2000

Generalized Multipartitioning

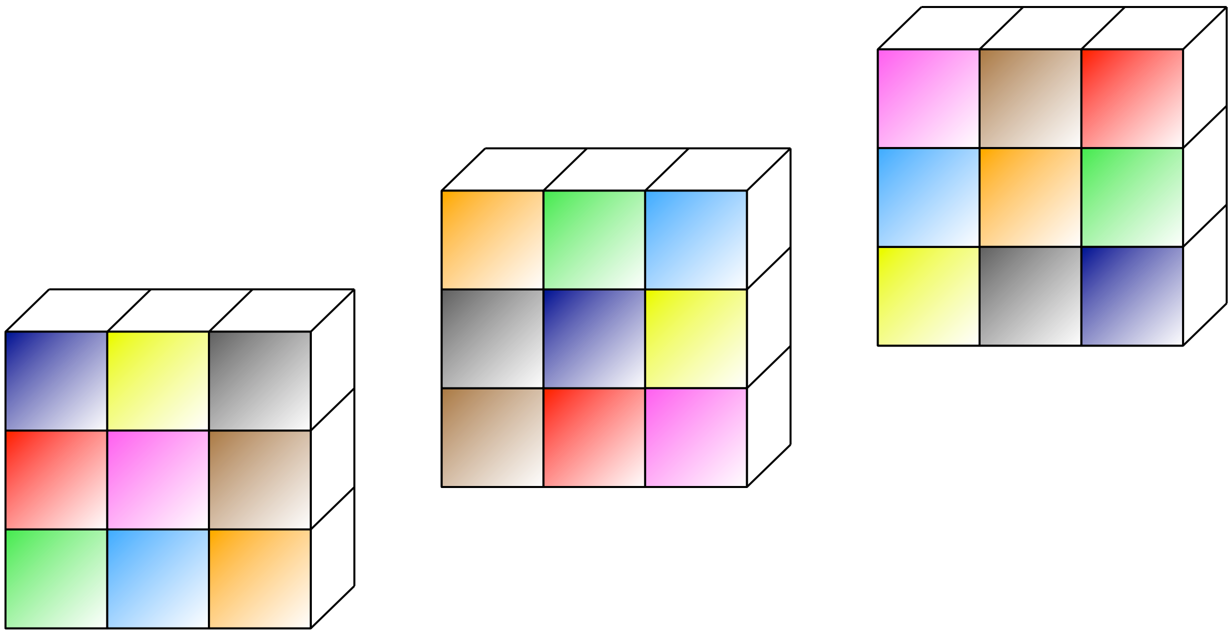
- Higher dimensional multipartitionings for arbitrary numbers of processors
  - Optimal overpartitionings (more than one tile per processor per hyperplane) + modular mappings
  - Compiler aggregates carried communication for hyperplanes



3x6x2

3D Multipartitioning for 6 processors

Higher-dimensional Multipartitioning



Diagonal 3D Multipartitioning for 9 processors