Simplifying Parallel Code Development

Alex Aiken
Stanford University

Joint work with Wonchan Lee, Manolis Papadakis, Elliott Slaughter

The Dream of Auto-Parallelization

```
for i in A:
  A[i] = f(i)

for i in B:
  B[i] = g(A[i+1])
```

Sequential Code

```
parallel for p = 0, P:
  int S = N/P                // N = |A|
  int lA[S+1], lB[S]

  for i = 0, S:
    lA[i] = f(p*S+i)

  if p > 1:   send(p-1,  lA[0])
  if p < P-1: recv(p+1, &lA[S])
  wait

  for i = 0, S:
    lB[i] = g(lA[i+1])
```

Distributed Code

- Convert data parallelism into SPMD with explicit communication and synchronization

- Many efforts over decades
  - E.g., HPF, SUIF, ...

Programming with First Class Partitions

Provide synchronization and communication from **multiple data partitions**

```
// pA1 is a partition of A
parallel for x in pA1:
  sA = pA1[x]
  for i in sA:
    sA[i] = f(i)
```

Implicit communication and synchronization
(for every `i` and `j` such that `pA1[i]` ∩ `pA2[j]` ≠ ∅)
→ Handled automatically by runtime system

```
// pA2 is another partition of A
parallel for x in pB:
  sA = pA2[x]
  sB = pB[x]
  for i in sB:
    sB[i] = g(sA[i+1])
```

- Configurability via runtime APIs (e.g., mapping API in Legion)
- Performance via efficient, scalable runtime implementations

Auto-Parallelization as Constraint Satisfaction

Auto-parallelization amounts to finding **legal partitions** that satisfy **partitioning constraints**

```
parallel for x in pA1:
  sA = pA1[x]
  for i in sA:
    sA[i] = f(i)

parallel for x in pB:
  sA = pA2[x]
  sB = pB[x]
  for i in sB:
    sB[i] = g(sA[i+1])
```

Find partitions **pA1**, **pA2**, and **pB** that satisfy these **constraints**:

1. **pA1** covers **A**
2. **pB** covers **B**
3. For any index `i` in `pB[x]`, `pA2[x]` includes `i+1`

Partitions can be **provided by users** or **synthesized by a solver**

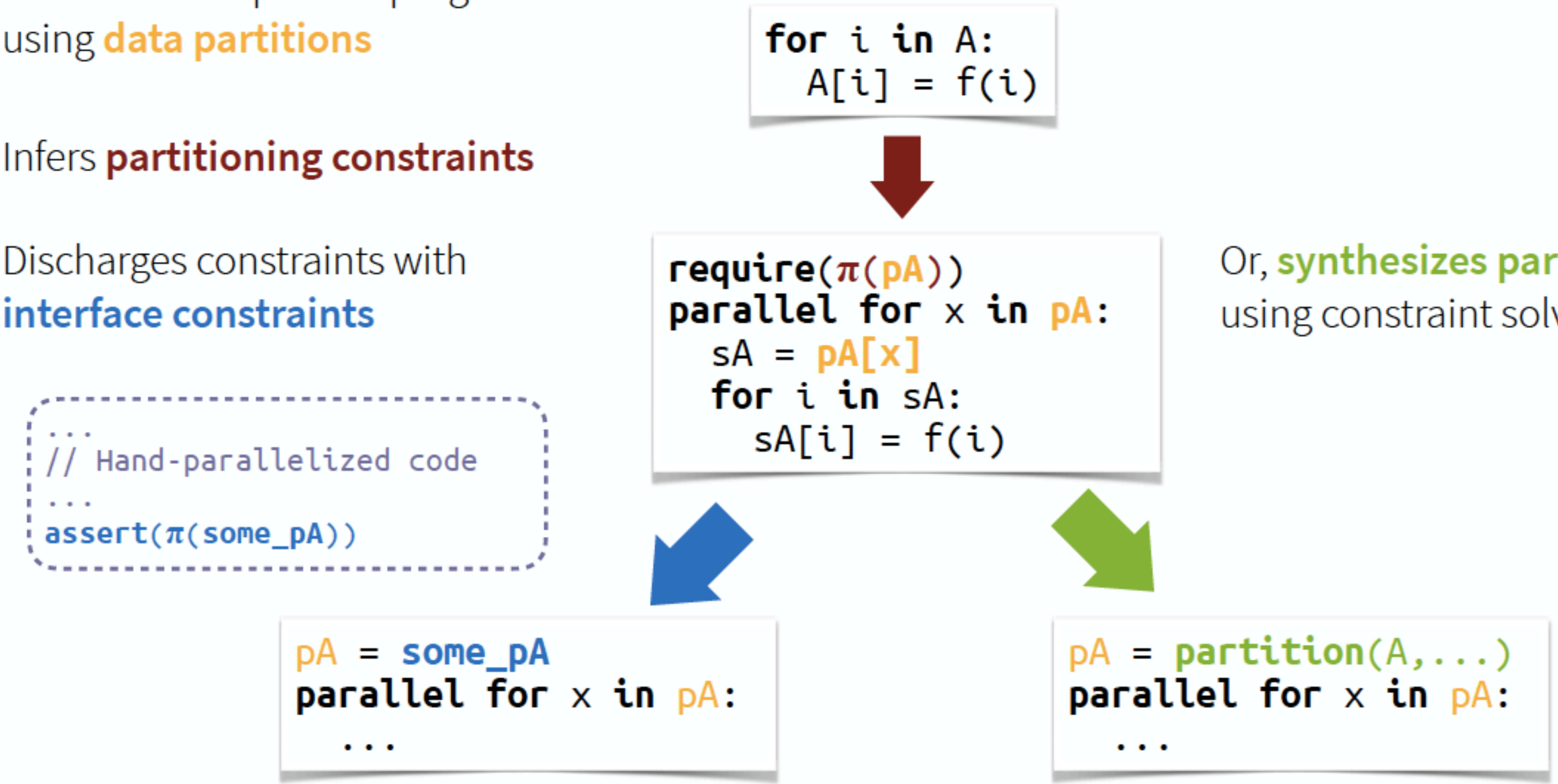
Overview

Parallelizes sequential program using **data partitions**

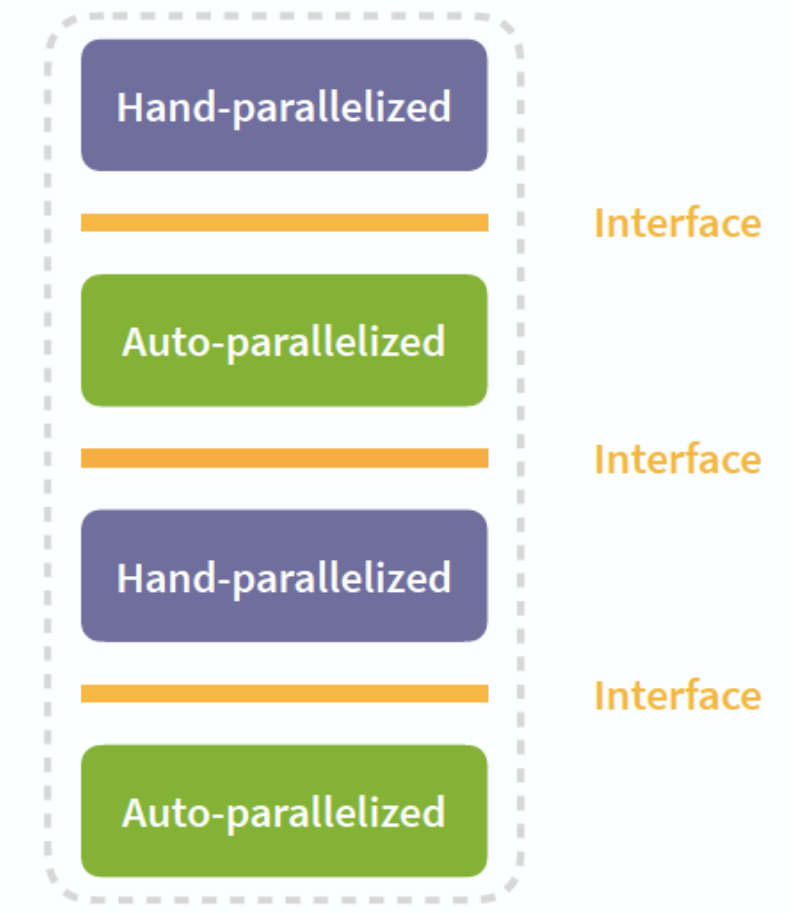
Infers **partitioning constraints**

Discharges constraints with **interface constraints**

Or, **synthesizes partitioning code** using constraint solver

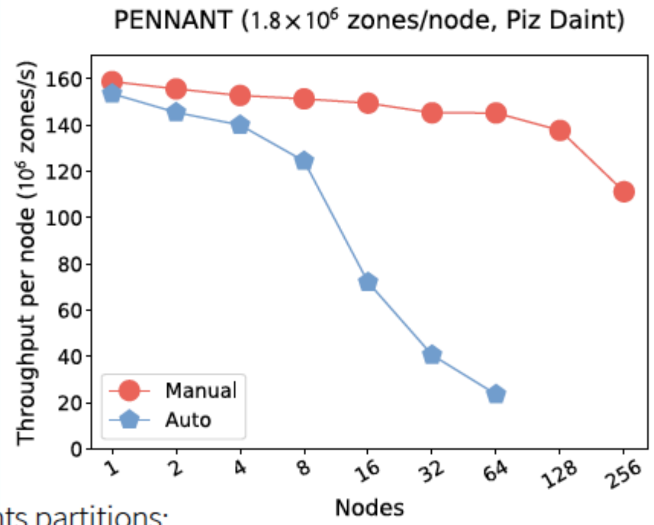


What’s Better?

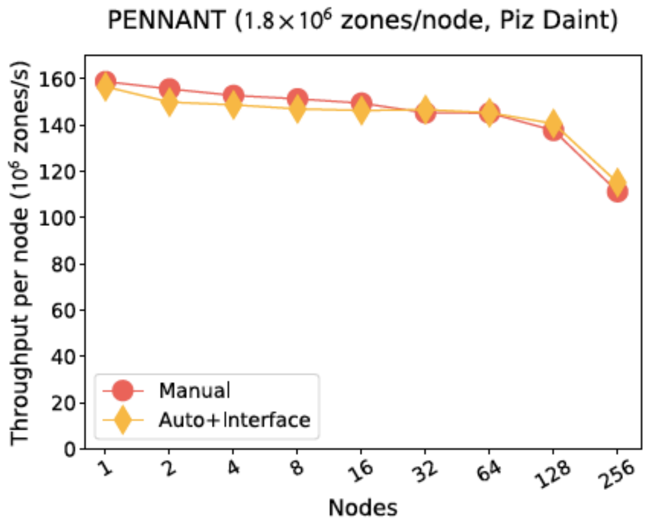


- Constraints provide an interface language
- Makes it much easier to mix auto- and hand-parallelized components
- Allows synthesized code to make explicit requirements for surrounding code
- Allows users to control synthesis by imposing external constraints

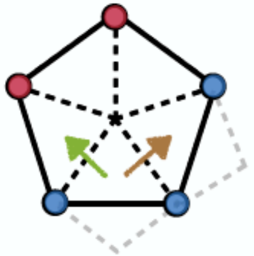
Example: Parallelization of Pennant



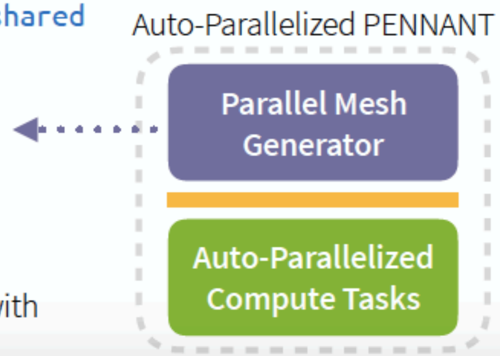
Add interface constraints



Two points partitions:
pPoints_private and pPoints_shared



Each side of a polygon colocates with its previous and next sides



Interface constraints:

$\texttt{complete(pPoints\_private} \cup \texttt{pPoints\_shared,Points)}$

$\texttt{image(pSides,Sides[}\cdot\texttt{].prev\_side)} \subseteq \texttt{pSides}$

$\texttt{image(pSides,Sides[}\cdot\texttt{].next\_side)} \subseteq \texttt{pSides}$

Summary

- Auto-parallelization
  - First-class partitions + partitioning constraints
  - Resulting constraint satisfaction problem gives a wide range of correct strategies for parallelization
- Interfaces
  - Constraints also provide a natural way to convey additional information to the synthesizer
  - To interoperate with hand-parallelized code
  - To exploit additional domain-specific knowledge