An Integrative Model for Parallelism

Victor Eijkhout
ICERM workshop

2012/01/09

- Introduction
- Formal part
- Examples
- Extension to other memory models
- Conclusion

Introduction

Practical starting point

- Data movement is very important: need to model this
  I'm starting with distributed memory.
- We understand parallelism through MPI (running at 300,000 cores); the problem is one of programmability.
- *Goal* To find a high level way of parallel programming, that can be compiled down to what I would have written in MPI
  - high productivity because working in global terms
  - portable performance since the MPI code is derived
- *Pie in the sky* Apart from distributed memory, we would like to cover task parallelism, multicore, accelerators

My claim

- It is possible to have a global model of parallelism where MPI-type parallelism can be derived through formal reasoning about global objects.
- The model is versatile in dealing with applications and memory models beyond linear algebra on clusters.
- Prospects for a high productivity high performance way of parallel programming.
- No silver bullet: algorithm designer still needs to design data and work distributions.

Claim in CS terms

Given distributions on input, output, instructions:
a communication pattern becomes a formally derived first-class object;
communication is an instantiation of this object

Consequences:

- communication is properly aggregated
- data can be sent (asynchronously) when available
- inspector-executor model where the pattern is derived once, then applied many times.

Working title:

I/MP: Integrative Model for Parallelism

Formal part

Theoretical model

Computation is bipartite graph from input to output, edges are instructions

$$A = \langle \mathrm{In}, \mathrm{Out}, E \rangle$$

Edge:

$$(\alpha, \beta) \in E: \quad \alpha \in \mathrm{In}, \beta \in \mathrm{Out}$$

is elementary computation between data elements.

This models a kernel, (not a whole code), possibly even less: compare 'supersteps' in BSP

Parallel computation:

$$A = \langle A_1, \ldots, A_P \rangle, \qquad A_p = \langle \mathrm{In}_p, \mathrm{Out}_p, E_p \rangle,$$

where

$$\mathrm{In} = \bigcup_p \mathrm{In}_p, \mathrm{Out} = \bigcup_p \mathrm{Out}_p, E = \bigcup_p E_p;$$

- Possibly non-disjoint partitions (redundant work)
- Example. If all $\mathrm{In}_p, \mathrm{Out}_p$ identical: shared memory
- Note explicit distribution of work, independently of data

Where does communication come from?

Define total input/output of a processor:

$$\begin{aligned}\mathrm{In}(E_p) &= \{\alpha \colon (\alpha,\beta) \in E_p\},\\ \mathrm{Out}(E_p) &= \{\beta \colon (\alpha,\beta) \in E_p\}.\end{aligned}$$

In_p	local part of input
$\mathrm{In}(E_p)$	total needed input
Out_p	local part of output
$\mathrm{Out}(E_p)$	total computed output

Not the same as $\mathrm{In}_p, \mathrm{Out}_p$!

- Communication: $\mathrm{In}(E_p) - \mathrm{In}_p$ is communicated in before computation
- Communication: $\mathrm{Out}(E_p) - \mathrm{Out}_p$ is communicated out after computation
- Example. Owner computes: $\mathrm{Out}(E_p) = \mathrm{Out}_p$
- Example. Embarrassingly parallel: $\mathrm{In}(E_p) = \mathrm{In}_p$, $\mathrm{Out}(E_p) = \mathrm{Out}_p$, and everything disjoint
- *By defining the distributions, communication becomes formally derivable*

Object view of communication

- Analyze all computation edges in terms of p, q source/target processors
- The sum over all p, q of these is an object describing the total communication in a kernel
- Compare to a PETSc VecScatter object:
  describe what goes where, not how.
- Compare to active messages: communication and computation in one.
- *Communication objects are derivable from the algorithm, they are not defined during the execution of the communication; inspector/executor paradigm*

Examples

What am I going to show you?

- The algorithm is mathematics
- Data distributions are easy to design
- *The communication is derived through formal reasoning*
- Some distributions lead to efficient communication, some don't: we still need a smart programmer.
- Today I'm only showing your math, but I think it's implementable

Example: Matrix-vector product

In_p	local part of input
$\mathrm{In}(E_p)$	total needed input
Out_p	local part of output
$\mathrm{Out}(E_p)$	total computed output

$$\forall_i : y_i = \sum_j a_{ij} x_j.$$

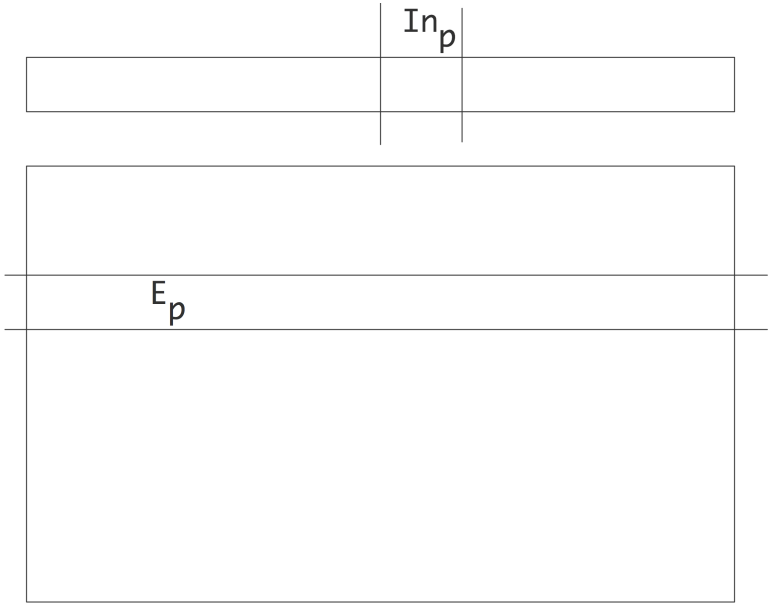
Split computation in temporaries and reduction:

$$\begin{aligned} \forall_i : y_i &= \textstyle\sum_j a_{ij} x_j \\ &= \textstyle\sum_j t_{ij}, \quad t_{ij} = a_{ij} x_j. \end{aligned}$$

Work distribution: t_{ij} is computed where a_{ij} lives
We will derive communication for row/col-distributed matrix

MVP by rows

In_p	local part of input
$\mathrm{In}(E_p)$	total needed input
Out_p	local part of output
$\mathrm{Out}(E_p)$	total computed output



Data:

$$\mathrm{In}_p = x(I_p)$$

work:

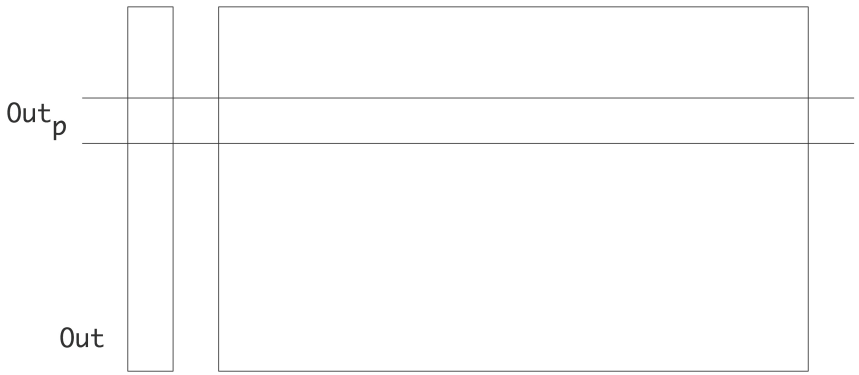
$$\begin{cases} E_p = \{\tau_{ij} : i \in I_p\}, \\ \tau_{ij} := \text{`}\, t_{ij} \leftarrow a_{ij} x_j \,\text{'} \end{cases}$$

derived fact:

$$\mathrm{In}(E_p) = x(*)$$

$\Rightarrow$ allgather

MVP by rows



Data:

In_p	local part of input
$\mathrm{In}(E_p)$	total needed input
Out_p	local part of output
$\mathrm{Out}(E_p)$	total computed output

$$\mathrm{In}_p = t(I_p, *)$$

work:

$$\begin{cases} E_p = \{\eta_i : i \in I_p\} \\ \eta_i := \text{`}\, y_i \leftarrow \sum_j t_{ij} \,\text{'} \end{cases}$$

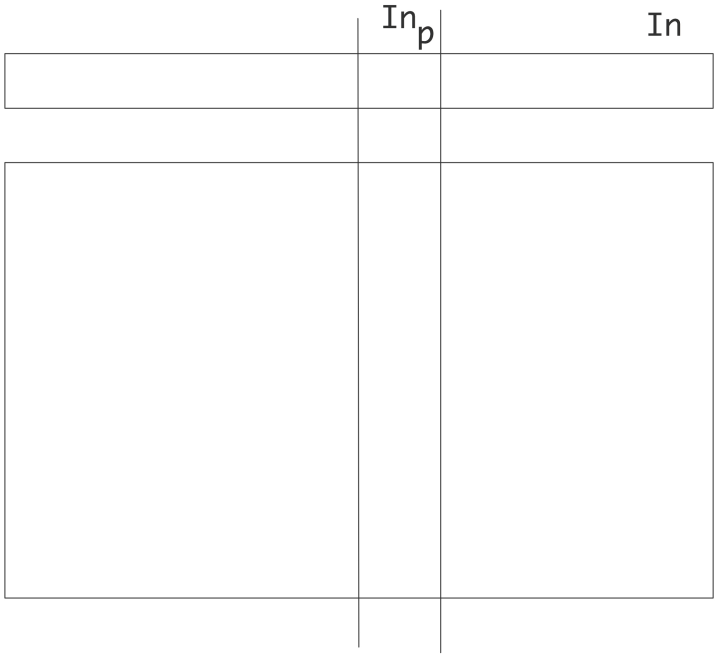
derived fact:

$$\mathrm{In}(E_p) = t(I_p, *)$$

$\Rightarrow$ local operation

MVP by columns

In_p	local part of input
$\mathrm{In}(E_p)$	total needed input
Out_p	local part of output
$\mathrm{Out}(E_p)$	total computed output



t_{ij} calculation:

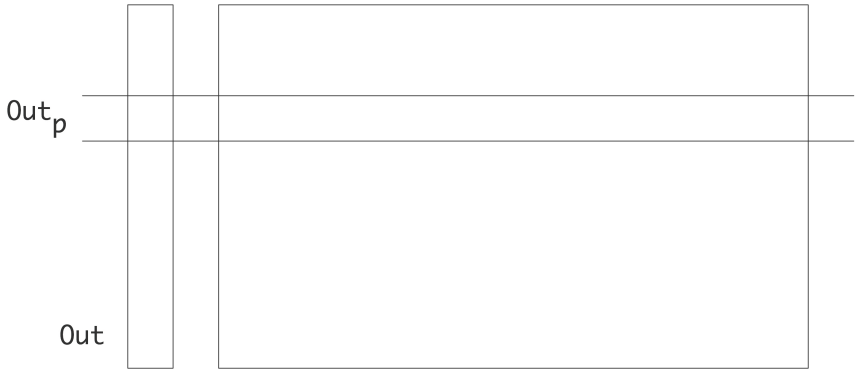
$$\mathrm{In}(E_p) = x(I_p), \quad \mathrm{In}_p = x(I_p)$$

$\Rightarrow$ local

$$\mathrm{Out}(E_p) = t(*, I_p)$$

MVP by columns

In_p	local part of input
$\mathrm{In}(E_p)$	total needed input
Out_p	local part of output
$\mathrm{Out}(E_p)$	total computed output



reduction:

$$\mathrm{In}(E_p) = t(I_p, *)$$

but now derived:

$$\mathrm{In}_p = t(*, I_p)$$

$\Rightarrow$ global transpose or reduce-scatter

Sparse MVP

A little more notation, but no essential difference

LU solution

Derivation of messages is identical, now no longer data parallel communication.

VecPipeline instead of VecScatter, dataflow

Note: IMP kernels do not imply synchronization:
dataflow, supply driven execution

Example: one IMP kernel per level in an FFT butterfly
operations execute based on availability of data
(a bit like DAG scheduling but not entirely)

Complicated example: n-body problems

Tree computation framework according to Katzenelson 1989:

- The field due to cell i on level ℓ is given by

$$g(\ell, i) = \bigoplus_{j \in C(i)} g(\ell + 1, j)$$

  where $C(i)$ denotes the set of children of cell i

- The field felt by cell i on level ℓ is given by

$$f(\ell, i) = f(\ell - 1, p(i)) + \sum_{j \in I_\ell(i)} g(\ell, j)$$

  $p(i)$: the parent cell of i; $I_\ell(i)$: interaction region of i

Implementation

$$E^{(g)} = E^{\tau} \cup E^{\gamma}, \quad E^{\tau} = \{\tau_{ij}^{\ell}\}, \quad E^{\gamma} = \{\gamma_i^{\ell}\}$$

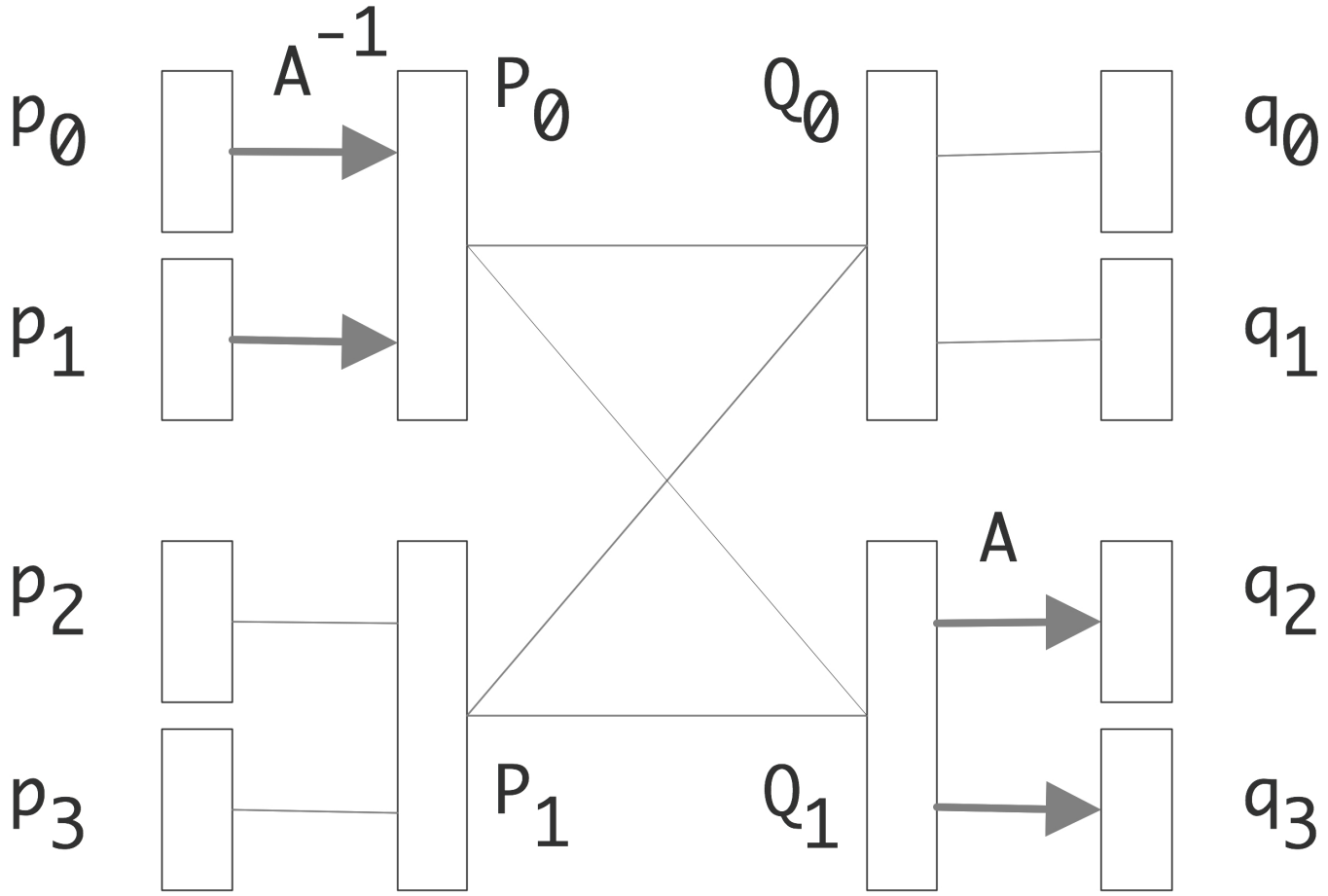
$$\begin{cases} \forall_i \forall_{j \in C(i)}: & \tau_{ij}^{\ell} = \text{`} \; t_{ij}^{\ell} = g_j^{\ell+1} \; \text{'} \\ \forall_i: & \gamma_i^{\ell} = \text{`} \; g_i^{\ell} = \oplus_{j \in C(i)} t_{ij}^{\ell} \; \text{'} \end{cases}$$

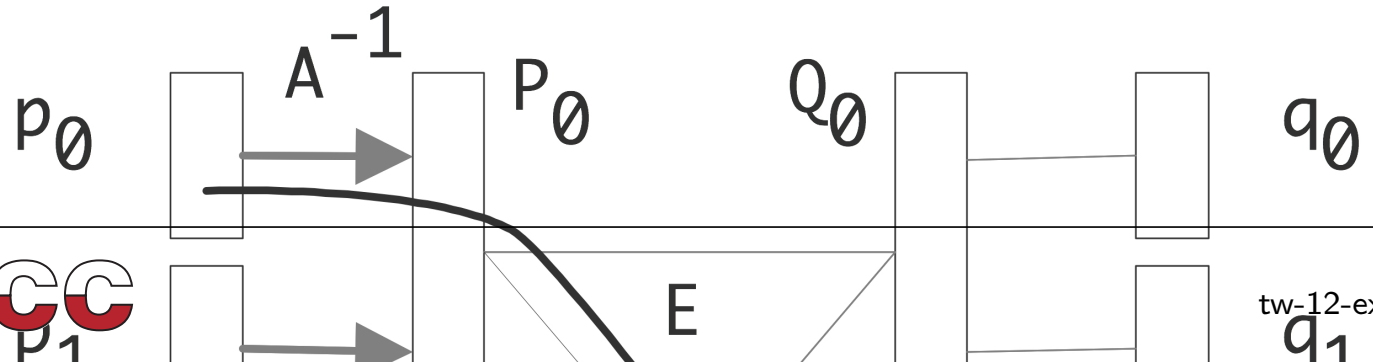
$$E^{(f)} = E^{\rho} \cup E^{\sigma} \cup E^{\phi}, \quad E^{\rho} = \{\rho_i^{\ell}\}, \quad E^{\sigma} = \{\sigma_{ij}^{\ell}\}, \quad E^{\phi} = \{\phi_i^{\ell}\}$$

$$\begin{cases} \forall_i: & \rho_i^{\ell} = \text{`} \; r_i^{\ell} = f_{p(i)}^{\ell-1} \; \text{'} \\ \forall_i \forall_{j \in I_{\ell}(i)}: & \sigma_{ij}^{\ell} = \text{`} \; s_{ij}^{\ell} = g_j^{\ell} \; \text{'} \\ \forall_i: & \phi_i^{\ell} = \text{`} \; f_i^{\ell} = r_i^{\ell} + \sum_{j \in I_{\ell}(i)} s_{ij}^{\ell} \; \text{'} \end{cases}$$

Extension to other memory models

Attached accelerators





Decompose GPU-GPU operations E as

$$e = A^{-1} \star e' \star A$$

where A is memory copy from CPU to GPU, so

$$e' = A \star e \star A^{-1}$$

is between CPUs: quotient graph.

We have derived the MPI communication of an algorithm with distributed GPUs through formal means

Shared memory multicore

- Processors still operate on local stores (L1/L2 cache)
- So it's really distributed memory
- except with the 'shared' memory as network
- again factor out physical data movement from logical data movement
- (this still needs work)

Conclusion

Conclusion

- Integrative Model for Parallelism
- also: Interface without Message Passing
- Based on explicit distribution of everything that needs distributing
- Communication objects derived through formal reasoning
- Trivially covers distributed memory, can be interpreted for other models
- Theoretical questions remain: correctness, deadlock, relation to DAG models, interpretation as functional model, relation to BSP, suitability for graph algorithms
- Practical question: how do you implement this? (Inspiration from PETSc, ParPre, Charm++, elemental)