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

\[ A = \langle \text{In}, \text{Out}, E \rangle \]

Edge:

\[ (\alpha, \beta) \in E : \quad \alpha \in \text{In}, \beta \in \text{Out} \]

is elementary computation between data elements.

This models a kernel, (not a whole code), possibly even less: compare ‘supersteps’ in BSP
Parallel computation:

\[ A = \langle A_1, \ldots, A_P \rangle, \quad A_p = \langle \text{In}_p, \text{Out}_p, E_p \rangle, \]

where

\[ \text{In} = \bigcup_p \text{In}_p, \quad \text{Out} = \bigcup_p \text{Out}_p, \quad E = \bigcup_p E_p; \]

- Possibly non-disjoint partitions (redundant work)
- Example. If all \( \text{In}_p, \text{Out}_p \) identical: shared memory
- Note explicit distribution of work, independently of data
Where does communication come from?

Define total input/output of a processor:

\[ \text{In}(E_p) = \{ \alpha: (\alpha, \beta) \in E_p \} , \]
\[ \text{Out}(E_p) = \{ \beta: (\alpha, \beta) \in E_p \} . \]

Not the same as \( \text{In}_p, \text{Out}_p \)!

- Communication: \( \text{In}(E_p) - \text{In}_p \) is communicated in before computation
- Communication: \( \text{Out}(E_p) - \text{Out}_p \) is communicated out after computation
- Example. Owner computes: \( \text{Out}(E_p) = \text{Out}_p \)
- Example. Embarrassingly parallel: \( \text{In}(E_p) = \text{In}_p, \text{Out}(E_p) = \text{Out}_p, \) 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

\[ \forall i : y_i = \sum_j a_{ij} x_j. \]

Split computation in temporaries and reduction:

\[ \forall i : y_i = \sum_j t_{ij}, \quad t_{ij} = a_{ij} x_j. \]

Work distribution: \( t_{ij} \) is computed where \( a_{ij} \) lives

We will derive communication for row/col-distributed matrix
MVP by rows

Data:

\[ \text{In}_p = x(l_p) \]

work:

\[
\begin{align*}
E_p &= \{ \tau_{ij} : i \in l_p \}, \\
\tau_{ij} &:= 't_{ij} \leftarrow a_{ij}x_j'
\end{align*}
\]

derived fact:

\[ \text{In}(E_p) = x(*) \]

⇒ allgather
MVP by rows

Data:

\[ \text{In}_p = t(I_p, \ast) \]

work:

\[ \begin{cases} E_p = \{ \eta_i : i \in I_p \} \\
\eta_i := 'y_i \leftarrow \sum_j t_{ij}' \end{cases} \]

derived fact:

\[ \text{In}(E_p) = t(I_p, \ast) \]

⇒ local operation
MVP by columns

$t_{ij}$ calculation:

\[ \text{In}(E_p) = x(l_p), \quad \text{In}_p = x(l_p) \]

\[ \Rightarrow \text{local} \]

\[ \text{Out}(E_p) = t(*, l_p) \]
MVP by columns

In\( (E_p) = t(I_p, \ast) \)

but now derived:

\[ \text{In}_p = t(\ast, I_p) \]

⇒ 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 $\ell$ 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 $\ell$ 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

\[ E(g) = E^\tau \cup E^\gamma, \quad E^\tau = \{\tau_{ij}^l\}, \quad E^\gamma = \{\gamma_i^l\} \]

\[
\begin{align*}
\forall i \forall j \in C(i) : & \quad \tau_{ij}^l = 't_{ij}^l = g_j^{l+1}' \\
\forall i : & \quad \gamma_i^l = 'g_i^l = \bigoplus_{j \in C(i)} t_{ij}^l'
\end{align*}
\]

\[ E(f) = E^\rho \cup E^\sigma \cup E^\phi, \quad E^\rho = \{\rho_i^l\}, \quad E^\sigma = \{\sigma_{ij}^l\}, \quad E^\phi = \{\phi_i^l\} \]

\[
\begin{align*}
\forall i : & \quad \rho_i^l = 'r_i^l = f_{p(i)}^{l-1}' \\
\forall i \forall j \in l_k(i) : & \quad \sigma_{ij}^l = 's_{ij}^l = g_j^l' \\
\forall i : & \quad \phi_i^l = 'f_i^l = r_i^l + \sum_{j \in l_k(i)} s_{ij}^l'
\end{align*}
\]
Extension to other memory models
Attached accelerators

\[ p_0 \xrightarrow{A^{-1}} p_0 \quad P_0 \xrightarrow{A} Q_0 \]

\[ p_1 \xrightarrow{A^{-1}} p_0 \quad P_0 \xrightarrow{A} Q_0 \]

\[ p_2 \xrightarrow{A} Q_1 \]

\[ p_3 \xrightarrow{A^{-1}} p_0 \quad P_0 \xrightarrow{A} Q_0 \]

\[ q_0 \quad q_1 \quad q_2 \quad q_3 \]
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)