Parallel Programming for the 21st Century
(Integrative Parallel Programming in HPC)

JARA HPC Symposium 20161004

Victor Eijkhout

2016
• Introduction

• Motivating example

• Formalism

• Demonstration

• Looking forward to Exascale

• Conclusion

• Backup slides: CS-y stuff
Introduction
Scientific algorithms vs parallel programming

- We want to express algorithms, and algorithms are sequential:
- Power method: matrix time vector, norm of vector, scale vector by norm, repeat...
- The only parallelism comes from having distributed objects: this is much more restricted than general parallelism
  *Generalized data parallelism*
- So why can’t we program that way?
  Sequential code on distributed objects.
  *sequential semantics*, see HPF
- Can we formalize sequential semantics so that compiler/runtime can give efficient execution?
Proposal: Integrative Model for Parallelism

- Theoretical model for describing parallelism in ‘generalized data parallel’ case
- Coding independent of hardware parallelism
- Implemented as library
- Minimal, yet sufficient, specification of parallel aspects
- Many aspects are formally derived, including messages and task dependencies.
IMP programming

- Top level: user constructs algorithm out of sequence of ‘kernels’ (matrix-vector product, inner product, et cetera)
- Inspector-executor: explicit invocation of analysis create a dataflow representation
- ... and execution, possibly with non-obvious optimizations.
Distributions

Distributions have been used before: mapping data to processors.

Wrong:

1. ad hoc: why are arrays distributed, but scalars not?
2. hard to express overlapping or redundant distributions.

Better: map processors to data.
Distribution example

• Grid coarsening: start with 6 points on 4 processors
• Coarse grid of 3 points on 4 processors involves duplication.
• Coarsening operation is local on some processor, not on others.
• Coarse grid is dynamically defined.
Motivating example
Three-point averaging

Data parallel function evaluation:

\[ y_i = f(x_i, x_{i-1}, x_{i+1}) \]

Algorithm structure:
each output index depends on three input indices
Computation structure

Define: task \equiv \text{computation on one processor:}

Each task has a dependency on three tasks
(dependency is MPI message or OpenMP task wait)

Can we derive this communication/synchronization structure from the algorithm structure?
$\alpha, \gamma$ distributions

Distribution: for each process a set of indices not necessarily disjoint

- $\alpha$ distribution: data assignment on input
- $\gamma$ distribution: data assignment on output

Specified by the user.
\( \beta \) distributions

Distribution \( \beta \): for each processor the elements it needs for the computation

- This is normally considered ‘halo’ (but includes local data)
- We consider it a parallel data distribution,
- ... just with occasional duplications.

Beta distribution incorporates data and algorithm properties.
Dataflow

We get a dependency structure: which processors (tasks) contribute to $\beta_p$?

Dependency $q \rightarrow p$ if

$$\alpha(q) \cap \beta(p) \neq \emptyset$$

Having the $\beta$-distribution is enough to derive communication/synchronization
Algorithm

By putting together these dependencies from many operations (‘kernels’) we get a dataflow graph.

Dependencies can be realized as MPI messages or OpenMP task waits.

Coding is independent of actual parallelism: same source code works as message passing, shared memory DAG, or hybrid.
Kernel structure vs task structure

Example: heat equation, Programmed structure vs derived
Big question at this point

How do you specify enough about the algorithm to make the dataflow derivable?
Formal definition of distributions

Distribution: mapping from processors to sets of indices. 

*opposite from usual!*

Given indices $N = \{0, \ldots, N - 1\}$
and processors $P = \{0, \ldots, P - 1\}$,
distribution:

$$u: P \rightarrow 2^N$$
Examples of distributions

Let $\beta = N/P$;

block:

$$b \equiv p \mapsto [p\beta, \ldots (p + 1)\beta - 1]$$

cyclic:

$$c \equiv p \mapsto \{ i : \text{mod} (i, \beta) = p \}$$

redundant:

$$\ast \equiv p \mapsto N$$
Operations on distributions

Let \( g : \mathbb{N} \to \mathbb{N} \), extend to operations on sets:

\[
\begin{align*}
g : 2^\mathbb{N} &\to 2^\mathbb{N} \\
g(S) &\equiv \{g(i) : i \in S\}
\end{align*}
\]

and operations on distributions:

\[
\begin{align*}
g : (P \to 2^\mathbb{N}) &\to (P \to 2^\mathbb{N}) \\
g(d) &\equiv \forall p \, d(p) \mapsto g(d(p))
\end{align*}
\]
Examples of operations on distributions

- Shift: $g: i \mapsto i + c$ ‘stencil’ language
- Coarsening: if $d$ is fine grid distribution, then $d/2$ is coarse grid.
Signature function

Describes the data parallel algorithm structure:

\[ y_i = f(x_i, x_{i-1}, x_{i+1}) \]
\[ \sigma(i) = \{ i, i - 1, i + 1 \} \] \[ \Rightarrow \{ y_i = f(x_{i_0}, \ldots, x_{i_k-1}) \] \[ \sigma(i) = \{ i_0, \ldots, i_{k-1} \} \]
Theoretical result

\[ \beta = \sigma_f \gamma \]

‘The beta distribution can be derived from the signature of the function and the output distribution’

- User needs to specify signature function
  Note: this is an algorithm property; nothing to do with parallelism
- User specifies parallel distribution of input and output.
- Communication/synchronization is formally derived from \( \alpha, \beta \) distributions:
  no user involvement needed.
Demonstration
Heat equation kernel

IMP_decomposition *decomp =
    new IMP_decomposition(arch,....)
IMP_distribution *blocked =
    new IMP_block_distribution
    (decomposition,globalsize);

IMP_object *input_vector = new IMP_object( blocked );
IMP_object *output_vector = new IMP_object( blocked );

IMP_kernel *update_step =
    new IMP_kernel(input_vector,output_vector);
update_step->add_sigma_oper( new ioperator(">>1") );
update_step->add_sigma_oper( new ioperator("<<1") );
update_step->add_sigma_oper( new ioperator("none") );
update_step->set_localexecutefn( &threepoint_execute );
Analyze and execute

algorithm *heat;
heat = new IMP_algorithm(decomp);

... heat->add_kernel( initialize ); ...
heat->add_kernel( update_step ); ...

heat->analyze_dependencies();
heat->execute();
Function pointer to local code

%% imp_functions.cxx

void central_difference_damp
    (int step, processor_coordinate *p,
     std::vector<object*> *invectors, object *outvector,
     double damp, double *flopcount)
{

double *outdata = outvector->get_data(p), *indata = invectors->get_data(p);

multi_indexstruct *pstruct = outvector->get_processor_structure(p);

domain_coordinate pfirst = pstruct->first_index_r(), plast = pstruct->last_index_r();

index_int lo=pfirst[0], hi=plast[0];

for (index_int i=lo; i<hi; i++) {

    index_int
        Iin = INDEX1D(i, in_offsets, in_nsize),
        Iout = INDEX1D(i, out_offsets, out_nsize);

        *damp;

}
Indexing independent of parallelism

General indexing:

- Global domain
- Numa domain
- Process domain

MPI indexing:

- Global domain
- MPI: Numa domain = process domain

Indexed point

Local index

Global index

offset
Almost like shared memory programming

- This uses global indexing.
- Same code works for MPI/OpenMP.
- The magic is in the offsets: boilerplate.
FE element/node mapping

Map element node to global (assembled) node:

```
for( lnode=0 ; lnode<8 ; ++lnode )
{
    Index_t gnode = elemNodes[lnode];
    x_local[lnode] = mesh.x(gnode);
    y_local[lnode] = mesh.y(gnode);
    z_local[lnode] = mesh.z(gnode);
}
```

Typical solution: indirection

Domain specific packages have tools for this.
**FE mapping: signature**

Signature function

\[ \sigma(g) = (2 \ast g - \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}, \ldots 2 \ast g) \]

Implementation:

```c
%% lulesh_functions.cxx
multi_indexstruct *signature_local_from_global
    ( multi_indexstruct *g,multi_indexstruct *enc ) {
    int dim = g->get_same_dimensionality(enc->get_dimensionality());
    domain_coordinate_allones allones(dim);
    multi_indexstruct *range =
        new contiguous_multi_indexstruct( g->first_index_r()*2-allones,g->last_index_r()*2 );
    return range->intersect(enc);
};
```
FE mapping: local code

```cpp
%% lulesh_functions.cxx
for (index_int i=pfirst[0]; i<=plast[0]; i++) {
  bool skip_first_i = i==out_gfirst[0], skip_last_i = i==out_glast[0]
  for (index_int j=pfirst[1]; j<=plast[1]; j++) {
    bool skip_first_j = j==out_gfirst[1], skip_last_j = j==out_glast[1]
    outdata[ INDEX2D(i,j,out_offsets,out_nsize) ] =
      ( !skip_first_i && !skip_first_j
        ? indata[ INDEX2D(2*i-1,2*j-1,in_offsets,in_nsize) ] : 0 )
      +
      ( !skip_first_i && !skip_last_j
        ? indata[ INDEX2D(2*i-1,2*j,in_offsets,in_nsize) ] : 0 )
      +
      ( !skip_last_i && !skip_first_j
        ? indata[ INDEX2D(2*i,2*j-1,in_offsets,in_nsize) ] : 0 )
      +
      ( !skip_last_i && !skip_last_j
        ? indata[ INDEX2D(2*i,2*j,in_offsets,in_nsize) ] : 0 )
    ;
  }
}
else if (dim==1) {
  index_int itar = tar0, isrc = src0, global_last = outvector->global_volume();
  for (index_int g=outvector->first_index(p).coord(0);
      g<=outvector->last_index(p).coord(0); g++) {
    index_int e = g/2; int m = g%2;
    if (g>=2 && g<global_last-1) {
      if (m==0) {
        outdata[itar++] = indata[isrc] + indata[isrc+2]; isrc++;
      } else {
        outdata[itar++] = indata[isrc] + indata[isrc+2]; isrc += 3;
      }
    } else
      outdata[itar++] = indata[isrc++];
  }
  else
    throw(fmt::format("Can not sum_mod2 for dim={}" ,dim));
  *flopcount += outlen;
}
```
Fun distribution stuff: coarsening

ioperator *div2 = new ioperator("/2");
for (int nlevel=0; nlevel<nlevels; nlevel++) {
    if (nlevel==0) {
        distributions[0]
            = new mpi_block_distribution(decomp,global_size);
    } else {
        distributions[nlevel] =
            distributions[nlevel-1]->operate(divby2);
    }
    objects[nlevel] = new mpi_object(distributions[nlevel]);
}
Conjugate Gradient from ready-made kernels

```c
rrp = new IMP_object(scalar);
kernel *rrcopy = new IMP_copy_kernel( rr,rrp );
algorithm->add_kernel(rrcopy);

kernel *matvec = new IMP_centraldifference_kernel( pnew,q );
algorithm->add_kernel(matvec);

kernel *pap_inprod = new IMP_innerproduct_kernel( pnew,q,pap );
algorithm->add_kernel(pap_inprod);

kernel *alpha_calc = new IMP_scalar_kernel( rr,"/",pap,alpha );
algorithm->add_kernel(alpha_calc);
```
This basically does the right thing.
Is this just about ease of programming?

- This looks a little like PETSc: one line per major operation. Difference: lines are not executed but added to algorithm structure; Inspector-executor: opportunity for algorithm re-organization.
- High level yet general, for instance no PETSc restrictions on distributions.
- No explicit mention of parallelism: same code can be executed MPI, shared memory DAG models, or hybrid.
- Analysis stage can rearrange algorithm: latency hiding.
Other applications

Partly working:

• N-body problems
• K-means clustering
• Dense linear algebra
Looking forward to Exascale
Directions

• MPI-only becomes MPI+X, X=threading? light-weight threads?
• Communication more expensive than work: communication minimization through duplicate work,
• Managed high-speed memory instead of cache (KNL): IMP ‘messages’ implemented as copy-to-scratchpad.
• Do we need a new hardware design? IMP can lead to software hardware co-design.
MPI+X

IMP implements MPI+DAG:
embedded distributions in each MPI process.
Tasks instead of processes

class product_block_distribution
  : public product_distribution, public block_distribution {
  public:
    product_block_distribution
    (decomposition *d, int o, index_int l, index_int g)
    : product_distribution(d), block_distribution(d, o, l, g),
      distribution(d) {
      embedded_distribution
      = new omp_block_distribution
      (omp_decomp, get_processor_structure( d->mycoord() ));
    };
Latency hiding and communication minimization

• Modern hardware can offload non-blocking messages to network card:
  (MPI 3 non-blocking collectives?) possibility for latency hiding
• Latency hiding intrinsic in algorithm, or through work replication:
  possible through overlapping distributions
• User realization tricky:
  – derive algorithm variants? (see Eijkhout, Bientinesi, van de Geijn 2010)
  – how early can you post i-operations?
  – how much buffer space does this take?
Overlapping kernels

Simple latency hiding through i-collectives:
Simultaneous matrix-vector product and inner product in CG variant

\[ z_2 = z_1 + M^{-1}q_1 \alpha \]

\[
\begin{align*}
\rho_2 &= r_2^t z_2 \\
\beta &= \rho_2 / \rho_1 \\
q_2 &= Az_2 + q_1 \beta
\end{align*}
\]

(Our kernels are partially ordered; the system detects possible overlap).
Communication avoiding

Classic block iteration with communication aggregation:

1. send halo
2. compute locally
3. compute missing values from halo

Demmel minimal communication scheme:

1. compute locally
2. send "halo" locally
3. finish computing locally
4. compute missing values from halo
Task graph subdivision and duplication

This scheme can be formally defined in IMP applied to task graphs.
Latency hiding in shared memory

- Asynchronous data offload ~ MPI iSend
- Use to place between KNL quadrants
- Use to ‘offload over fabric’ between Xeon and MIC.
Co-design speculation

• Current hardware is ‘Von Neumann’ architecture: driven by ‘data misses’.

• IMP supports a ‘push’ model which can simplify hardware design.
  – Asynchronous transfer to scratchpad, instead of cache-misses
  – Latency hiding and communication avoiding through early ‘Isend’ posting.

• Result: simpler processor design:
  – DRAM controller can do better scheduling
  – Processor needs less out-of-order scheduling:
    far more energy-efficient, at current technology levels.
Conclusion
Integrative Parallel Programming

- Theoretical point: separate specification of algorithm and data is enough for deriving parallel communication/synchronization.
- Practical point: sequential programming of parallel algorithms
- Possibility of
  - Mode-independent programming
  - Automatic performance improvements
  - Easy adding functionality such as load balancing
  - Coding of unexpected algorithms (1.5D, 2.5D)
Backup slides: CS-y stuff
Functional model

We have sort of a functional programming system: a kernel produces an object.

There is no concept of overwriting data

⇒ no race condition

Updates

\[ y_i = f(x_{i_1}, \ldots, x_{i_k}) \]

are supposed to be atomic, well-defined, hidden in function pointer, so not dependent on parallelism
Functional model’

There is no concept of overwriting data
So how do you deal with garbage collection?

MPI: message sent, then buffer safe to reuse
OpenMP: there is no ‘acknowledgement of read’
Looped reasoning in closed task graph

A task graph is *closed* if:

\[ \forall v \forall v' \in \text{succ}(\text{pred}(v)) : \text{succ}(v) \cap \text{succ}(v') \neq \emptyset \]
Theorem

For any task $p$ in a closed task graph, if it has an active successor, that is,

$$\exists s \in \text{succ}(p): A(s)$$

then all its predecessors are concluded:

$$\forall q \in \text{pred}(p): C(q).$$