TACC Technical Report IMP-26

Load Balancing in IMP

Victor Eijkhout*

May 25, 2018

* eijkhout@tacc.utexas.edu, Texas Advanced Computing Center, The University of Texas at Austin

Abstract

Discussion of load balancing.

The following IMP reports are available or under construction:

IMP-00 The IMP Elevator Pitch
IMP-01 IMP Distribution Theory
IMP-02 The deep theory of the Integrative Model
IMP-03 The type system of the Integrative Model
IMP-04 Task execution in the Integrative Model
IMP-05 Processors in the Integrative Model
IMP-06 Definition of a 'communication avoiding' compiler in the Integrative Model (under construction)
IMP-07 Associative messsaging in the Integrative Model (under construction)
IMP-08 Resilience in the Integrative Model (under construction)
IMP-09 Tree codes in the Integrative Model
IMP-10 Thoughts on models for parallelism
IMP-11 A gentle introduction to the Integrative Model for Parallelism
IMP-12 K-means clustering in the Integrative Model
IMP-13 Sparse Operations in the Integrative Model for Parallelism
IMP-14 1.5D All-pairs Methods in the Integrative Model for Parallelism (under construction)
IMP-15 Collectives in the Integrative Model for Parallelism
IMP-16 Processor-local code (under construction)
IMP-17 The CG method in the Integrative Model for Parallelism (under construction)
IMP-18 A tutorial introduction to IMP software (under construction)
IMP-19 Report on NSF EAGER 1451204.
IMP-20 A mathematical formalization of data parallel operations
IMP-21 Adaptive mesh refinement (under construction)
IMP-22 Implementing LULESH in IMP (under construction)
IMP-23 Distributed computing theory in IMP (under construction)
IMP-24 IMP as a vehicle for software/hardware co-design, with John McCalpin (under construction)
IMP-25 Dense linear algebra in IMP (under construction)
IMP-26 Load balancing in IMP (under construction)
IMP-27 Data analytics in IMP (under construction)

1 Theory

We use the *diffusion* model of load balancing [1, 2]:

- processes are connected through a graph structure, and
- in each balancing step they can only move load to their immediate neighbours.

This is easiest modeled through a directed graph. Let ℓ_i be the load on process i, and $\tau_i^{(j)}$ the transfer of load on an edge $j \to i$. Then

$$\ell_i \leftarrow \ell_i + \sum_{j \to i} \tau_i^{(j)} - \sum_{i \to j} \tau_j^{(i)}$$

Although we just used a i, j number of edges, in practice we put a linear numbering the edges. We then get a system

$$AT = \bar{L}$$

where

- A is a matrix of size $|N| \times |E|$ describing what edges connect in/out of a node, with elements values equal to ± 1 depending;
- T is the vector of transfers, of size $|E|$; and
- $\bar{L}$ is the load deviation vector, indicating for each node how far over/under the average load they are.

In the case of a linear processor array this matrix is under-determined, with fewer edges than processors, but in most cases the system will be over-determined, with more edges than processes. Consequently, we solve

$$T = (A^t A)^{-1} A^t \bar{L} \qquad \text{or} T = A^t (A A^t)^{-1} \bar{L}.$$

Since $A^t A$ and $A A^t$ are positive indefinite, we could solve the approximately by relaxation, needing only local knowledge.

2 Implementation

Our basic tool is the `distribution_sigma_operator`. Most operators take a `multi_indexstruct` and transform it to a new one; this operator can take the whole distribution into consideration, which allows us to do things like averaging.

```
%% balance_functions.cxx
distribution *transform_by_average(distribution *unbalance,double *stats_data) {
  if (unbalance->get_dimensionality()!=1)
    throw(std::string("Can only average in 1D"));
```

```
if (!unbalance->is_known_globally())
  throw(fmt::format
("Can not transform-average <<{}>>: needs globally known",unbalance->get_name()));

decomposition *decomp = dynamic_cast<decomposition*>(unbalance);
if (decomp==nullptr)
  throw(std::string("Could not cast to decomposition"));

auto astruct = new parallel_structure(decomp);
int nprocs = decomp->domains_volume();
for (int p=0; p<nprocs; p++) {
  auto me = unbalance->coordinate_from_linear(p);
  double
    cleft = 1./3, cmid = 1./3, cright = 1./3,
    work_left,work_right,work_mid = stats_data[p];
  index_int size_left=0,size_right=0,
    size_me = unbalance->volume(me);

  if (p==0) {
    size_left = 0; work_left = 0;
    cleft = 0; cmid = 1./2; cright = 1./2;
  } else {
    size_left = unbalance->volume( me-1 );
    work_left = stats_data[p-1];
  }



  if (p==nprocs-1) {
    size_right = 0; work_right = 0;
    cright = 0; cmid = 1./2; cleft = 1./2;
  } else {
    size_right = unbalance->volume( me+1 );
    work_right = stats_data[p+1];
  }

  index_int new_size = ( cleft * work_left * size_left + cmid * work_mid * size_me
 + cright * work_right *size_right ) / 3.;

  auto idx = std::shared_ptr<indexstruct>( new contiguous_indexstruct(1,new_size) );
  auto old_pstruct = unbalance->get_processor_structure(me);
  auto new_pstruct = std::shared_ptr<multi_indexstruct>
    ( new multi_indexstruct( std::vector<std::shared_ptr<indexstruct>>{ idx } ) );
  astruct->set_processor_structure(me,new_pstruct);
}
```

```
  astruct->set_is_known_globally();
  return unbalance->new_distribution_from_structure(astruct);
}
```

Unfortunately, this being integer calculation, we lose a couple of elements, and so we have another operator that stretches a distribution to a preset size.

```
%% template_balance.cxx
auto average =
  new distribution_sigma_operator
  ( [load] (distribution *d,processor_coordinate *p) -> std::shared_ptr<multi_indexstruct> {
    return transform_by_average(d,p,load); } );
  newblock = block->operate(average,true);
```

3 Example

We test an synthetic benchmark based on

- a two-dimensional mesh, distributed over a two-dimensional processor grid, with
- work per grid point that is modeled by a time-dependent bell curve

$$w(x,t) = 1 + e^{-(x-s(t))^2} \quad \text{where } s(t) = tN/T.$$

This model is encountered in practice with, for instance, weather codes where the adaptive local time integration is strongly location-dependent. Our benchmark does not perform actual work: each processor evaluates the total amount of work for its subdomain and sleeps for a proportional amount of time.

After each time step we evaluate the total time per processors, and perform a 'diffusion load balancing' step [1, 2]. While this, strictly speaking, optimizes for the previous time step, not the next, if the load changes slow enough this will still give a performance improvement.

Indeed, we see up to 20 percent improvement in runtime.

Procs:	64	320	672
Balanced runtime:	63	192	856
Unbalanced:	72	266	935

4 Discussion

In this report we have shown how the Integrative Model for Parallelism (IMP) programming model makes it possible to have dynamically changing data distributions,

in particular where the changes are dictated by application demands. We have given a proof of concept of this by realizing a dynamic load balancing scheme completely in user space, not relying on any system software support. Our tests show that the overhead of recomputing distributions and recreating data does not negate the performance enhancement of the improved distributions.

References

[1] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7(2):279–301, October 1989.

[2] Y. F. Hu and R. J. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25:417–444, 1999.