User Level Load Balancing in the Integrated Model for Parallelism

Victor Eijkhout*

December 2, 2019

Abstract

We describe how the Integrative Model for Parallelism makes it possible to implement load balancing in user spaces. IMP has distributions as first class, dynamically created objects, so that it becomes possible to create new distributions based on dynamic conditions such as load. Load balancing can then be interpreted as a copy operation between a distributed object, and the same object on a new distribution.

By way of demonstration, we implement the 'diffusion' algorithm for load balancing and show improvement of runtime on a synthetic problem.

1 Introduction

Load balancing is the act of, at runtime, changing the assignment of work to processing elements. In a shared memory context, this is easily done by dynamic assignment of tasks. However, in a distributed memory context the distribution work is tightly bound to the distribution data. While an even distribution of data is often feasible, this may not always result in an even distribution of work. For instance, if the local work involves integration of stiff Ordinary Differential Equations (ODEs), the relation between data and work can be hard to predict, and in fact dynamically involving. Thus, runtime adjustments of the assignment of data to processors may be needed.

Implementing load balancing is difficult since it relies on many low level assumptions on how the problem is structured. Consider the case of Partial Differential Equations (PDEs). In the usual implementation, each process stores a sparse matrix corresponding to the equations on the local domain. Load distribution then

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

implies moving rows from one processor to another. Typical sparse matrix data structures do not allow such dynamic restriction and extension. The next problem is that such changes to the matrix may alter the communication structure.

For purposes of this discussion we are abstracting away from these matters and only consider dynamic updates to the partitioning of the domain of definition of the PDE. We then assume that based on this new partitioning the local coefficient matrices can easily be reconstructed. We also assume that the communication structure is not changed by the redistribution.

1.1 The Integrative Model for Parallelism

The Integrative Model for Parallelism (IMP) is a new development in parallel programming, based on a mathematical formalization of data distributions. The theory is descsribed in detail in [**?**]. Here we give a brief outline.

The basic concept of IMP is that of distributions, which describe the correspondence between data an processing elements. Objects are then constructed using these distributions. Traditionally, distributions are assignments of data to processors, but in IMP we turn this around: we map processors to data. This means that we can have overlapping or redundant distributions.

The next aspect of IMP is that of data parallel functions $y \leftarrow f(x)$: computations where each component y_i of the output y is an independent calculation of components $x_{i_1} \ldots x_{i_k}$ of the input x. The signature function σ then describes this mapping from output indices to input:

$$\sigma(i) = \{i_1, \ldots, i_k\}.$$

In many cases this signature function is easy to describe; for instance multigrid coarsening has a signature function (in 1D) of

$$\sigma(i) = \{2i, 2i+1\}.$$

Finally we introduce the 'beta-distribution' of the operation: if γ is the distribution of y, then $\beta = \sigma(\gamma)$.

The value of these concepts is that, with α the distribution of x, we have the lemma that for any pair p, q of processes

$$\text{if } \alpha(q) \cap \beta(p) \neq \emptyset \text{ then there is a dependency from process } p \text{ to } q.$$

Summarizing, task dependencies, including in the form of MPI messages, can be derived from the data distributions and the algorithm signature function.

The aspect of IMP that concerns us here is that distributions are dynamically created objects, defined in user space. We argue that this makes it possible to

implement load distribution, under the above assumptions: based on measured runtimes we can decide to adjust the data assignment in such a way to more load away from overloaded processes. This means that a new distribution is created based on the dynamic evaluation of local runtimes.

> The redistribution of a PDE domain of definition can be formulated as a copy operation between an object (such as PDE coefficients) on two different distributions: the current one, and a dynamically constructed one that purportedly gives a more equitable assignment of load to processors.

1.2 Outline

The rest of this paper is devoted to a presentation how a common load balancing algorithm can easily be implemented in IMP. We will show tests on a synthetic benchmark example, showing that this scheme is able to track a moving peak in workload, and redistribute data accordingly.

2 Diffusion load balancing

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

- 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

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

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

3 Implementation

In this section we sketch the implementation of the distribution transforming mechanism. A precise explanation would require including the full reference manual of the IMP library [**?**], so we hope the reader will indulge us and use their imagination for terms that are left undefined. We use a top-down presentation.

The `distribution` class has a method `operate` that yields a new distribution. Thus:

```
block = block->operate(diffuse);
```

The `diffuse` operator is a so-called `distribution_sigma_operator`, which is a derived class of a more basic operator type.

```
auto diffuse =
    distribution_sigma_operator
    ( [stats_vector,adjacency,trace,mytid]
      (shared_ptr<distribution> d) -> shared_ptr<distribution>  {
      return transform_by_diffusion(d,stats_vector,adjacency);  }
    );
```

The only function that is completely written in user terms is `transform_by_diffusion`, which is about 60 lines long, but most of those are concerned with extracting information from the original distribution, and constructing the new distribution from transformed data.

- Unpack original distribution.

```
/*
 * Input:
 *     shared_ptr<distribution> unbalance : original distribution
 * Output:
 *     vector<index_int> partition_points
 */
auto partition_points = unbalance->partitioning_points();
```

- Solve for the load move amounts

```
/*
 * Input:
 *     vector<double> times : runtimes per processor
 * Output:
 *     VectorXd loadmove    : amount of data to be moved
 * Method:
 *     use Eigen3 library to solve normal equations.
 */
VectorXd
  imbalance = VectorXd::Constant(nsegments,-avg_time),
  loadmove;
for (int it=0; it<ntimes; it++)
  imbalance[it] += times[it];

auto normal_matrix = adjacency * adjacency.transpose();
auto ata_fact = normal_matrix.ldlt();
auto balance_solve = ata_fact.solve( imbalance );
loadmove = adjacency.transpose() * balance_solve;
```

- Calculate new partitioning points and return distribution. This depends on the type of allowable distribution. One-dimensional code looks like:

```
/*
 * Input:
 *     new vector of local domain sizes
 * Output:
 *     new distribution
*/
parallel_structure shifted(decomp);
shifted.create_from_local_sizes(localsizes);
return unbalance->new_distribution_from_structure(shifted);
```

4 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 [**?**, **?**]. 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

5 Discussion

In this report we have shown how the 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] Victor Eijkhout. IMP code repository, 2014-7. https://bitbucket.org/VictorEijkhout/imp-demo.

[3] Victor Eijkhout. A mathematical formalization of data parallel operations. *CoRR*, abs/1602.02409, 2016. http://arxiv.org/abs/1602.02409.

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