# Reactive Rebalancing for Scientific Simulations running on ExaScale High Performance Computers

Roel WUYTS [a,d], Karl MEERBERGEN [b,d] Pascal COSTANZA [c,d]
[a] *imec, Leuven (Belgium)*
[b] *K.U.Leuven, Computer Science, Leuven (Belgium)*
[c] *Intel (Belgium)*
[d] *Intel ExaScience Lab Leuven (Belgium)*

**Abstract.** Exascale computers, the next generation of high performance computers, are expected to process 1 exaflops around 2018. However the processor cores used in these systems are very likely to suffer from unpredictable high variability in performance. We built a prototype general-purpose reactive work rebalancer that handles such performance variability with low overhead. We did an experimental validation by developing a reactive rebalancer library in UPC, and using it in a 5-point stencil (heat) simulation. The experiments show that our approach has very limited overhead that compensates for runtime processor speed variations, with or without simulated processor slowdowns.

**Keywords.** High performance computing, exascale computing, load balancing, UPC

## Introduction

ExaScale computing systems, the next generation high-performance computing systems will offer speeds of up to 1 exaflops (or $10^{18}$ floating-point operations per second). They are expected to become available by 2018. Many applications that will need this infrastructure will be scientific simulations, such as next-generation hurricane modeling, high-resolution climate models, De novo genome assembly [1] or space weather prediction. Exascale systems will likely consist of millions of cores executing applications with billions of threads, which are orders of magnitude bigger than existing HPC systems.

Given the timeframe they will likely use 12nm or smaller CMOS technology, according to the ITRS roadmap[1]. Preliminary studies in technology scaling indicate that processing elements built on this technology will exhibit high variability in performance. Moreover dynamic power management at the hardware level inside a CPU will result in performance variability between cores and across different runs [4].

[1] http://www.itrs.net/

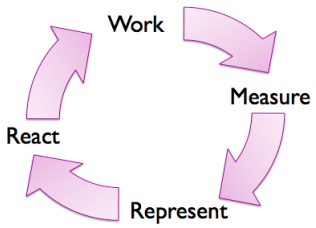



**Figure 1.** The four steps carried out every iteration in a rebalancing loop.

This high variability in performance will not be handled well by existing software packages like Zoltan[2], ParMetis[3], or Jostle[4]. In these packages dynamic load balancing means 'restoring work load balance while minimizing future communication costs' and requires data repartitioning and data migration. A trade-off must be found between the cost of the load balancer, the quality of the resulting data distribution, and the amount of data migration [6]. This is done by modeling the computational and communication costs for a particular decomposition of an application in threads/processes [7,8,2]. However this process implicitly assumes that the performance of the underlying hardware remains relatively constant. When the hardware exhibits high variability in performance this assumption no longer holds.

This paper proposes to tackle high variability in performance through a reactive work rebalancer that handles performance variability transparently and at very low cost. The rebalancer is distributed and memory-affinity aware, guiding the application in moving data when needed. It offers applications a *rebalancing loop* that iterates four steps, depicted in Figure 1: the loop body is executed, its performance is measured, the measurement is represented internally, and a decision is taken about what and how much rebalancing is needed. These four steps can be customized by applications:

- the work carried out in the loop that requires reactive rebalancing.
- what is measured, such as execution speed or the number of particles.
- the representation of the measurement, e.g. average or maximum value.
- the decision strategy that determines the reaction.

Since the performance variability of cores is unpredictable the rebalancer relies on the runtime measurements to determine whether the performance is drifting, and reacts to this if needed. Therefore we call the rebalancer *reactive*.

Note that the work rebalancer is not meant to replace application-specific load balancing schemes meant to compensate application-specific imbalances. Its goal is to deal with hardware variability.

We have implemented our work balancing approach in Unified Parallel C and used it in a simple 5-point stencil (heat) simulation. The experimental results show two things:

[2]http://www.cs.sandia.gov/zoltan/
[3]http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview
[4]http://staffweb.cms.gre.ac.uk/ wc06/jostle/

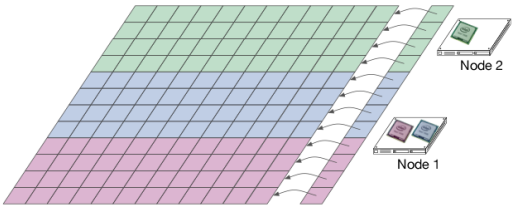



**Figure 2.** The grid data structure of our heat equation simulation in UPC.

1. the overhead of our rebalancer is small. When not simulating slowdowns the performance of using or not using rebalancing are similar.
2. when processors slow down, work is transferred from slower to faster processors.

The rest of the paper gives more details about the concept and implementation of the work rebalancer and about the experiments we carried out.

## 1. Implementing the Rebalancer and Application

For our experiments we have decided to implement the rebalancer library and an application in UPC (Unified Parallel C) [5], a fairly mature PGAS (partitioned global address space) extension of C. UPC is a SPMD language[5] that explicitly exposes a two-level memory hierarchy consisting of local memory and shared memory. Portions of the shared memory space may have an affinity for a particular thread (typically because it is physically stored on the machine where that thread runs), thereby exploiting locality of reference. Locks and barriers can be used for synchronization.

The application we have used to benchmark with, and which we also use in the remainder of this section when showing concrete examples, is a 5-point stencil (heat) simulation with explicit time stepping. The basic kernel consists of a loop that executes a stencil operation over a grid, reading values from one grid to calculate and writing values in another (they are swapped after every iteration).

We implemented the grid in UPC as a shared array of pointers to rows, where each row is again a shared array. This is shown in Figure 2 for a grid of twelve by twelve elements distributed over three UPC threads running on two compute nodes. The whole grid is accessible by all threads that make up the UPC program, but each thread has data affinity to a consecutive number of rows of the grid. Which threads have affinity to which rows is governed by the rebalancer, which starts out with an equal distribution by default, as shown in the figure.

In the heat simulation the work for a thread is defined as the processing of a consecutive number of rows. When needed, the rebalancer will therefore decide to adapt the number of rows that have to be processed by a thread to match the performance of other threads.

[5] *single program multiple data*, meaning that essentially the same program runs on all nodes

```
void rebalancing_loop(int start, int nr, timestamp (*step) ) {
  for (iteration = start; iter < start+nr; iteration ++) {
    upc_barrier;

    //potentially rebalance rows between threads
    rebalance(mgr, iteration);
    upc_barrier;
    adjustWork(mgr, *old, *new, iteration);
    incrementIterationCounter(mgr);

    //do a heat step and measure how long it takes
    upc_barrier;
    timeTaken = (*step)(mgr, iteration, *old, *new);

    //update the internal information
    update(mgr, iteration, timeTaken);
  }
}
```

**Figure 3.** The rebalancing loop (argument list truncated for space reasons)

The UPC rebalancing library offers applications a *rebalancing loop* construct, that takes as arguments the body that needs to be executed and a callback function. The latter has to ensure that each thread continues to process data for which it has affinity even after rebalancing. An example for the 5-point stencil simulation is shown in Listing 4. Note that this callback function cannot be part of the rebalancing library because the library is application independent.

The rebalancing loop is shown in Listing 3. It implements the four-step cycle outlined before, using execution times to measure the workload for each thread. Other load measures could be used as well[6].

The timings are supplied by the application as return value of the function executed as body. The rebalancer keeps an average of the loop body timings for all iterations since the last rebalancing of rows for that thread. This is done in the *update* invocation at the end of the iteration. Rebalancing is decided upon in the *rebalance* function and performed in the *adjustWork* function. The reason for this splitting and the barrier between them is a UPC technicality we will not explain in this paper.

More important is to discuss the load balancing strategy used. For reasons of simplicity and scaling we implemented the following strategy:
*Every thread compares its work measurement with the one of its left neighbor (with wraparound at the ends). When the thread is at least 5 percent slower than its left neighbor, work is pushed from the current thread to its left neighbor. The amount of work pushed is half of what would make the threads execute as fast.*

Figure 5 illustrates this algorithm. If the performance of CPU 2 on Node 1 (that runs the thread processing the middle four rows of the grid) is halved, some

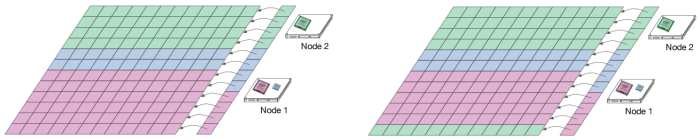
[6]For example, we also have a version that uses energy consumption information provided by the latest Intel Sandy Bridge processors.

```
void reassignRows(ThreadManager *mgr,
  SharedGrid old, SharedGrid new, int start_row, int end_row)
{
  row_p aRow;
  int i, j, row;

  for (row=start_row; row <= end_row; row++) {
    //allocate in my thread (affinity is my thread!)
    aRow = upc_alloc(old->size * sizeof(double));
    //copy data from thread with other affinity to me
    upc_memcpy(aRow, old->data[row], old->size * sizeof(double));
    //free old old row
    upc_free(old->data[row]);
    //install new row with affinity to me in the grid
    old->data[row] = aRow;

    //For the new grid, only copy top and bottom row.
    //the rest will be overwritten in the next iteration anyway.
    aRow = upc_alloc(new->size * sizeof(double));
    if (row == 0 || row == new->n)
      upc_memcpy(aRow, new->data[row], new->size * sizeof(double));
    upc_free(new->data[row]);
    new->data[row] = aRow;
  }
}
```

**Figure 4.** Example of a callback function specified by the application and called by the rebalancer when data actually has to be moved between threads to ensure data locality after rebalancing.




(a) 1 iteration after CPU 2 on node 1 slows down

(b) 2 iterations after CPU 2 on node1 slows down

**Figure 5.** Experimental Results.

rows will be moved left (Figure 5(a)). More work will then be rebalanced later (Figure 5(b)).

Several things have to be noted about this algorithm.

- the algorithm is deliberately kept very simple: threads only use information from a single neighbor. Approaches that use information from more threads, for example comparing against an 'average' value, are bound to cause bottlenecks on large systems.
- while consecutive threads all run within 5 percent difference with their neighbor, there is no global property enforced and so the difference between the slowest and fastest thread exceeds 5 percent (but is bound due to the wrap-around. This was again a trade-of between complexity of the

load balancing algorithm versus its results. The experiments show that our choice proved efficient, though alternatives could be envisioned.
- we only transfer half the amount of work when rebalancing. We did this to get a dampening effect. If the slowdown remains more work will be transferred in the consecutive steps.
- the strategy can easily be changed, since it is a separate component in our rebalancer. This allows for experimentation with various strategies.

## 2. Experimental Validation

The goal of our rebalancer is to handle variability in performance of processors. Therefore we implemented artificial straggling in our heat equation simulation: threads that straggle have a busy loop added that keeps the thread occupied. As a result the work done by that thread is taking more time, simulating that the processor running that tread runs slower. In our straggling experiments we double the processing time for a straggler thread, mimicking a processor that runs at 50 percent of its regular performance.

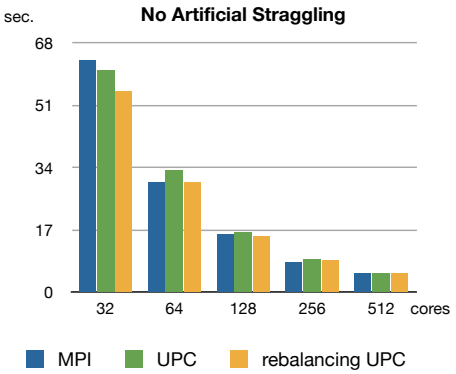
We ran our experiments on the *vic3* cluster of the Flemish Supercomputing Centre (VSC), using compute nodes featuring two Xeon 5650 "westmere" 2.66Ghz CPUs and 24GB of RAM, connected via Infiniband. Openmpi 1.4 (compiled with the Intel 64 Compiler 12.0.0.084) and Berkeley UPC 2.10.2 (compiled with the Intel 64 Compiler 1100.20090131 and using the *ibv* backend) were used.

All timings are the results of averaging the wall clock times of three runs of experiments. We show the results of running 200 time steps of the 5-point stencil "heat" simulation on a grid with size of 32768 by 32768 [7].

We performed three sets of experiments differing in the number of straggling threads.

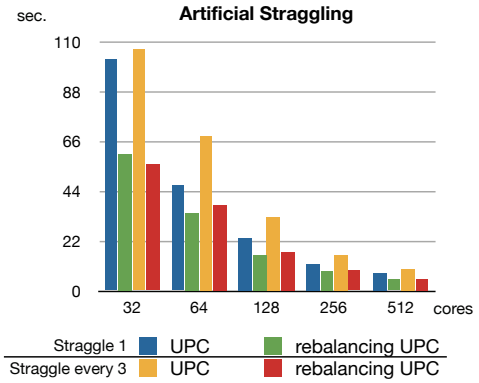
- no artificial straggling (Figure 6). In these experiments we do not add any straggling and compare an MPI implementation, a UPC implementation without rebalancing, and a UPC implementation with rebalancing. This shows the possible overhead of our approach when there is no obvious need to rebalance. The results show that there is almost no penalty paid when using the rebalancing library: it is sometimes a bit faster and sometimes a bit slower. The reason is that the rebalancing already compensates for minor performance differences on existing HPC systems, indicating that those could already benefit from rebalancing.
- one thread straggling (Figure 7, two left bars). In this experiment we straggle one thread. As can be expected the non-rebalancing version runs approximately twice as slow. The performance of the rebalancing version is close to the version without straggling.
- one third of the threads straggling (Figure 7, two right bars). In this experiment we straggle every third thread. Again the plain version takes a hit while the rebalancing version performs very well.

[7] We ran experiments with other grid sizes (from 128 to 16384) and numbers of time steps but because the conclusions are similar and because of space constraints we do not show them.




| | No Artificial Straggling | | |
|---|---|---|---|
| | MPI | UPC | rebalancing UPC |
| **32** | 63,30 | 60,57 | 54,79 |
| **64** | 30,02 | 33,30 | 30,13 |
| **128** | 15,86 | 16,53 | 15,43 |
| **256** | 8,21 | 9,04 | 8,87 |
| **512** | 5,21 | 5,21 | 5,27 |

**Figure 6.** Experimental results where we do not add any artificial straggling.




| | Straggle 1 | | Straggle every 3 | |
|---|---|---|---|---|
| | UPC | rebalancing UPC | UPC | rebalancing UPC |
| **32** | 102,19 | 60,52 | 106,64 | 55,83 |
| **64** | 46,64 | 34,38 | 68,53 | 38,16 |
| **128** | 23,57 | 16,09 | 32,73 | 17,26 |
| **256** | 11,89 | 8,80 | 16,09 | 9,11 |
| **512** | 8,02 | 5,13 | 9,62 | 5,41 |

**Figure 7.** Experimental results where we straggle a single thread and every third thread.

## 3. Future Work

First of all we are in the final steps of completing experiments with a particle-in-cell (PIC) simulation. In the PIC method, individual macro-particles in a Lagrangian frame which mimic the behavior of the distribution function are tracked in continuous phase space. Moments of the distribution function such as densities and currents are computed simultaneously on an Eulerian frame (fixed cells). More data structures are rebalanced in this implementation. Initial runs show similar results (but less dramatic), but more experiments are required to fully confirm the claims.

A second important part of research lies in the decision algorithm that governs the rebalancing. The current algorithm is very simplistic in nature. Even if it performs well in our experiments it does not do a global optimization. We there-

fore will implement other strategies known from load-balancing research that can infer global optimizations from local information and compare the results. We also want to analyze the convergence properties [3].

Thirdly we want to use a larger cluster to perform more scaling experiments.

## 4. Conclusion

The context of this work is exascale systems, the future high-performance systems. These systems will be subject to performance variability between cores and across different runs. Application-driven load balancing approaches currently used in high-performance computers will not be able to deal with this performance variability. In this paper we propose a reactive rebalancer that handles the high variability in performance through a reactive work rebalancer that measure work as it is executed, represents this information, and reacts to it according to a rebalancing strategy. The rebalancer is distributed with the application. It uses a callback mechanism so the application can continue to benefit from data locality even though work is rebalanced. We have implemented a library that implements a reactive rebalancer as well as a 5-point stencil simulation that uses it. Experiments show that our approach handles slowdowns very well, with an overhead that compensates for the cost of the rebalancing.

## Acknowledgments

This work is funded by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT), and by Intel.

## References

[1] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Ko, J. Levesque, D. A. Reed, V. Sarkar, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling. Exascale software study: Software challenges in extreme scale systems. Technical report, DARPA IPTO, September 2009.

[2] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007.

[3] G. V. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal on Parallel and Distributed Computing*, 7:279–301, 1989.

[4] J. Dongarra, P. Beckman, and et al. The international exascale software roadmap. *International Journal of High Performance Computer Applications*, 25(1):85, 2011.

[5] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. Wiley, 2005.

[6] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, pages 485–500, 2000.

[7] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[8] C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In F. Magoules, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007. (Invited chapter).