Supplementary Materials

Hi-Corrector: A fast, scalable and memory-efficient package for normalizing large-scale Hi-C data

Wenyuan Li, Ke Gong, Qingjiao Li, Frank Alber* and Xianghong Jasmine Zhou*

[1] Molecular and Computational Biology Program, Department of Biological Sciences, University of Southern California, Los Angeles, CA 90089, USA.

* Correspondence email: alber@usc.edu or xjzhou@usc.edu

This supplementary materials provide a detailed description of the algorithms and experiments.

Memory-Efficient Sequential and Parallel Algorithms

Given an observed chromosome contact frequency matrix $\mathbf{O}=(O_{ij})_{N\cdot N}$ over N genomic regions, the IC method eliminates biases so that all genomic regions have equal visibility (Imakaev *et al.*, 2012). We briefly summarize the iterative correction procedure in Figure S1. After the bias vector $\mathbf{b}=(b_i)_{N\cdot 1}$ is obtained, the corrected contact frequency matrix can be calculated as $\mathbf{T}_{ij}=O_{ij}/(b_i b_j)$. However, this implementation of the original IC algorithm needs to keep the whole matrix **O** in memory, in order to alternately update **b** and **O**.

To make this algorithm memory-efficient, we designed a strategy of breaking the matrix **O** into *K* equal partitions of complete rows and loading only one partition into memory at any given time. Therefore, the memory requirement can be very low when *K* is large. This strategy adapts the IC algorithm by adding two steps: (i) loading the *k*-th matrix partition O^k into memory and (ii) updating this partition with the last updated bias vector **b**. The new Memory Efficient Sequential algorithm (called IC-MES) works even for the extreme case of *K*=*N*, where only one row is loaded each time. The extra running time is the data loading time. Figure S2 shows the procedure of the IC-MES algorithm.

IC-MES is memory efficient, but it is still a sequential algorithm that runs on a single computing machine. Therefore, it may be too slow when the machine has small memory. To normalize large Hi-C matrices in a short time, we also designed a fast, scalable and Memory-Efficient Parallel algorithm (called IC-MEP) that can maximally exploit the parallelism of the normalization problem and make use of many commonly available computing resources.

In essence, the normalization problem is a data divisible task: a series of operations that can independently work on separate partitions of the data. This problem is perfectly suited to the data-parallel model in a distributed-memory computing environment such as a computer cluster, which consists of *K* independent processors (or nodes) that are loosely or tightly connected in high-speed networks and have limited local memory. We employed the manager-worker (or master-slave) parallel programming paradigm. The manager task partitions the data into *K* blocks, then initiates *K* worker tasks in different nodes; each worker task processes a single data block. The manager coordinates all workers and synchronizes their calculations with updated bias vectors. The IC-MEP algorithm has very little network messaging overhead, because no communication exists between workers. Therefore, it is computationally efficient. Furthermore, in order for each worker to run its task on limited memory, we also used the memory-saving strategy of the IC-MES algorithm. That is, each worker further partitions its

Input: the matrix **O**
Output: the bias vector **b**
Procedure: repeat the following steps in a limited number of iterations.

1. Initialize $\mathbf{b} = (b_i)_{N\cdot 1}$ to be a vector whose elements are all ones.
2. Perform the following steps in a limited number of iterations
   1) Compute the additional bias vector $\mathbf{t} = (t_i)_{N\cdot 1}$ based on **O**, where t_i is the element sum of the i-th row.
   2) Update **O**: $O_{ij} \leftarrow O_{ij}/(t_i * t_j)$ for all $i,j=1,\ldots N$.
   3) Update the bias vector **b**: $b_i \leftarrow b_i * t_i$, for all $i=1,\ldots, N$.

Memory used: N^2 for the whole matrix **O** and $2N$ for two vectors **b** and **t**

Figure S1. Original iterative correction algorithm (IC).

Input: the matrix **O**, number of partitions K
Output: the bias vector **b**
Procedure:

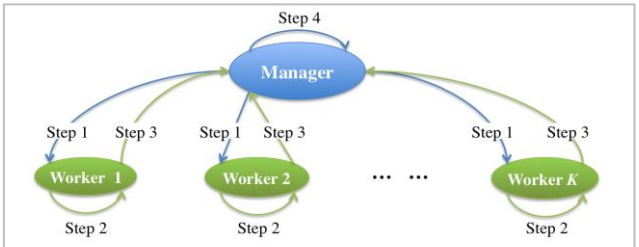
1. Equally partition **O** into K blocks; each block contains approximately N/K rows.
2. Initialize $\mathbf{b} = (b_i)_{N\cdot 1}$ to be a vector whose elements are all ones.
3. FOR each partition k from 1 to K
   1) Load the k-th block of C (denoted $\mathbf{O}^k$) into the memory
   2) Update $\mathbf{O}^k$: $\mathbf{O}^k(i,j) \leftarrow \mathbf{O}^k(i,j)/(b_i * b_j)$, where $\mathbf{O}^k(i,j)$ is the element O_{ij} and $i, j \in$ rows in the k-th block.
   3) Compute a temporary row sum: t_i, for $i \in$ rows in the k-th block
   4) Update b_i: $b_i \leftarrow b_i * t_i$, for only $i \in$ rows in the k-th block
4. Repeat Step 3 for each partition in a limited number of iterations.

Memory used: N^2/K for a matrix partition $\mathbf{O}^k$, N for the vector **b** and N/K for the temporary vector **t**

Figure S2. Memory-efficient sequential algorithm for iterative correction (IC-MES).

Input: the matrix **O**, number of partitions K
Output: the bias vector **b**
Procedure:

1. Partition **O** into K blocks; each block contains the same number of rows
2. Create one manager task and K worker tasks: each worker performs the computation for a block, while the manager coordinates all workers and synchronizes their computations with the updated bias vector. Below is the parallel schema.



Each iteration has 4 steps 1 — 4 as described below

Step 1: Synchronization
Manager notifies each worker to start another round of computation. In the 1st iteration, manager partition the whole data into K blocks and sends partition info to K workers.

Step 2: Computation
Each worker can further equally partition its assigned data to L sub-blocks. Then perform the following process for each sub-block.
1. Load the bias vector **b** into the memory from the file saved in the last iteration.
2. FOR each sub-block l from 1 to L
   1) Load the l-th sub-block $\mathbf{O}^l$ into the memory from the input data file
   2) Update $\mathbf{O}^l$: $O^l_{ij} \leftarrow O^l_{ij} / (b_i * b_j)$, where $i, j \in$ rows in the l-th sub-block.
   3) Compute row sums: t_i, for $i \in$ rows in the l-th sub-block
   4) Update b_i: $b_i \leftarrow b_i * t_i$, for only $i \in$ rows in the l-th sub-block

Step 3: Send to manager the updated **b**, and wait for the manager message of starting the next iteration.

Step 4: Collect updated **b** from all workers, then write **b** to a file

Memory used for each worker: $N^2/(KL)$ for a matrix sub-block $\mathbf{O}^l$ and $N/(KL)$ for a partition of the vector **b**.

Figure S3. Memory-efficient parallel algorithm for iterative correction (IC-MEP).

assigned data block into a set of sub-blocks and loads only one sub-block into memory at any given time. Therefore, theoretically the IC-MEP algorithm can work on any number of processors with any local memory capacity. Figure S3 details this parallel paradigm and the memory-efficient computation steps in each worker.

Implementations

We used ANSI C to implement the two sequential algorithms IC and IC-MES, because this language provides maximum control and efficiency over the memory used. We only used functions in the portable C library, so the code is easily portable to any operating system. We implemented the parallel algorithm IC-MEP using the popular Message Passing Interface (MPI), which is a highly standardized and portable environment designed for solving large-scale scientific problems on distributed memory systems. MPI has been widely implemented and installed in many types of distributed memory systems and even supercomputers. Our C and MPI implementation is also highly portable and computationally efficient, because we used only two of its most basic communication functions, MPI_Send and MPI_Recv, for passing a very small amount of data among processes.

Results

We compared three algorithms (IC, IC-MES, and IC-MEP) on the TCC/Hi-C data of two human cell types: GM12878 and hESC (Kalhor *et al.*, 2012; Dixon *et al.*, 2012). The whole genome is partitioned into the equal-size regions (or bins); the bin size is the main indicator of Hi-C data resolution. In these

experiments, we are interested in comparing the performance of normalizing contact data at the resolutions of 20K bp and 10K bp per bin.

The scalability and memory efficiency of IC-MEP were tested with different numbers of processors and memory sizes per processor. We used the high-performance computing (HPC) resource of the University of Southern California, which is a distributed-memory computing environment. Each node has a dual octocore (16 processors) Intel Xeon 2.4 GHz CPU and 128GB memory [1]. We also used these computers to run the sequential algorithm, IC-MES, on the single processor. The HPC system allows each job to limit the size of the memory used, so we can easily simulate different amounts of available memory for the memory efficiency tests.

In the experiment with 20K bp resolution data, the basic IC algorithm requires a minimum memory of 86GB. Using our algorithms IC-MES and IC-MEP, we can control the memory size for each processor to be 1GB or 4GB. The results are listed in Table S1. Although the basic IC algorithm can complete the normalization in about half hour, it needs too much memory. IC-MES can run with just 4GB memory (a common memory configuration in office computers) and complete the same work in reasonable time (within 4 hours). IC-MEP can dramatically speed up the computation by using more processors (about 6 minutes using 48 processors), while using only 1GB of memory in each processor. Please note that (i) the memory sizes reported here do not include memory used for system overhead such as the MPI environment; and (ii) the running time is dependent on the file I/O speed and the MPI system's efficiency.

For the 10K bp resolution data, none of the HPC computer nodes (with 128GB memory limit) can load the full matrix (about 343GB) for the basic IC algorithm. Table S1 shows that IC-MES and IC-MEP can use existing computing resources with common memory configurations to achieve the results with low memory usage and as quickly as possible.

Table S1. Running time of three algorithms (IC, IC-MES and IC-MEP) on 10K and 20K bp resolution Hi-C data for two human cell types, using different numbers of processors and amounts of memory. All algorithms were terminated after 10 iterations for the purpose of this performance comparison, since each iteration has almost the same running time. "Memory" includes only the memory allocated only for computation in each processor, not system overhead. The running time format is "hours : minutes : seconds".

	Algorithm	**IC**	**IC-MES**	**IC-MEP**	
20K bp data (151825 bins)	#Processor	1	1	16	48
	Memory	86GB	4GB	1GB	1GB
	Time (gm12878)	0:36:50	3:58:14	0:19:50	0:6:38
	Time (hESC)	0:35:01	3:49:18	0:19:48	0:6:47
10K bp data (303640 bins)	#Processor	1	1	16	48
	Memory	343GB	32GB	2GB	2GB
	Time (gm12878)	**NA**	47:27:32	4:50:03	0:26:02
	Time (hESC)	**NA**	37:26:15	4:49:27	0:26:09

References

Dixon,J.R. *et al.* (2012) Topological domains in mammalian genomes identified by analysis of chromatin interactions. *Nature*, **485**, 376–80.

Imakaev,M. *et al.* (2012) Iterative correction of Hi-C data reveals hallmarks of chromosome organization. *Nat. Methods*, **9**, 999–1003.

[1] http://hpcc.usc.edu/support/infrastructure/hpcc-computing-resource-overview/

Kalhor,R. *et al.* (2012) Genome architectures revealed by tethered chromosome conformation capture and population-based modeling. *Nat. Biotechnol.*, **30**, 90–8.