Parallel Computing

After briefly discussing the often neglected, but in praxis frequently encountered, issue of trivially parallel computing, we turn to parallel computing with information exchange. Our illustration is the replica exchange method, also called parallel tempering. Closely related (but less good) is the Jump-Walker (J-Walker) method.

Parallel tempering has a low, but non-trivial, communication overhead between the processes. Information about the temperatures, energies and nearest neighbor locations of n systems is communicated, and logical decisions have to be made. This algorithm gives some kind of ideal start into the world of parallel processing and is also of major practical importance. In some applications it competes with the multicanonical algorithm, in others both algorithms supplement one another, and combinations of them have been explored.

Trivially parallel computing

An example of a trivially parallel large scale endeavor is the *Folding@Home* distributed computing project, which achieved sustained speeds of about thirty teraflops in molecular dynamics simulations.

There are many tasks, which simply can be done in parallel, for instance canonical MC simulations at different temperatures and/or distinct lattices. For such applications the gain due to having many (identical) processors available scales linearly with the number of processors. In practice trivially parallel applications are frequently encountered. Another example are simulations of spin glass replica. It is far more cost efficient to run trivially parallel applications on PC clusters than on dedicated parallel machines (supercomputers), which feature expensive communication networks.

For MCMC simulations of single systems, which last (very) long in real time, parallelization is trivial, but there is some overhead due to multiple equilibration

runs. Typically the CPU time consumption of such a simulation is divided into

$$t_{tot} = t_{eq} + t_{meas} \ .$$

For parallelization of such a computation on n independent nodes the single processor CPU time scales as

$$t^1_{tot} = t_{eq} + t_{meas}/n \ .$$

How many processors should one choose (assuming a sufficiently large supply)? A good rule of thumb is

$$n = \texttt{nint} \left[\frac{t_{meas}}{t_{eq}} \right] \ .$$

The total processor time used becomes $t^n_{tot} = n\, t_{eq} + t_{meas} \approx 2\, t_{meas}$, which limits the CPU time loss due to multiple equilibrations to less than two

$$R_{loss} = \frac{t^n_{tot}}{t_{tot}} \approx \frac{2\, t_{meas}}{t_{eq} + t_{meas}} < 2 \ ,$$

and the gain is a reduction of the run time by the factor

$$R_{gain}^{-1} = \frac{t_{tot}^{1}}{t_{tot}} \approx \frac{2\,t_{eq}}{t_{eq} + n\,t_{eq}} = \frac{2}{1+n}\ .$$

The worst case is $n = 2$, for which the improvement factor in CPU time is $2/3$. Obviously this parallelization makes most sense for $t_{meas} \gg t_{eq}$.

Replica exchange method (parallel tempering)

The method of multiple Markov chains was introduced by Geyer and, independently, by Hukusima and Nemoto under the name **replica exchange** method. The latter work was influenced by the **simulated tempering**, which coined the name **parallel tempering** for an exchange of temperatures in multiple Markov chains. But not that simulated tempering is distinct method and actually a special case of the method of **expanded ensembles**. A precursor of such ideas is a 1986 PRL by Swendsen and Wang, where a general replica exchange method is formulated (apparently its generality prevented one from identifying the important special case).

In practice one does not exchange configuration but temperatures or similar parameters, so that the amount of exchanged information is small. Therefore, the method is particularly well suited for parallel processing on clusters of workstations, which lack the expensive fast communication hardware of the

dedicated parallel computers. The Jump Walker (J-Walker) approach does not exchange configurations, but feeds replica from a higher temperature into a simulation at a lower temperature, and has consequently problems with balance.

Parallel tempering performs n canonical MC simulations at different β-values with Boltzmann weight factors

$$w_{B,i}(E^{(k)}) = e^{-\beta_i E^{(k)}} = e^{-H} \ , \ \ i = 0, \ldots, n-1 \ ,$$

where $\beta_0 < \beta_1 < ... < \beta_{n-2} < \beta_{n-1}$, and allows to exchange neighboring β-values:

$$\beta_{i-1} \longleftrightarrow \beta_i \quad \text{for} \ \ i = 1, \ldots, n-1 \ .$$

Their joint weight is

$$e^{-\beta_{i-1}E_{i-1} - \beta_i E_i} = e^{-H}$$

and the $\beta_{i-1} \leftrightarrow \beta_i$ exchange leads to

$$
\begin{aligned}
-\triangle H &= \left(-\beta_{i-1}E_i^{(k)} - \beta_i E_{i-1}^{(k')}\right) - \left(-\beta_i E_i^{(k)} - \beta_{i-1}E_{i-1}^{(k')}\right) \\
&= (\beta_i - \beta_{i-1})\left(E_i^{(k)} - E_{i-1}^{(k')}\right)
\end{aligned}
$$

which is accepted or rejected according to the Metropolis algorithm, *i.e.*, with probability one for $\triangle H \leq 0$ and with probability $\exp(-\triangle H)$ for $\triangle H > 0$.

The CPU time spent on the β exchange should be less than 50% of the total updating time. In practice it is normally much less (a few percent). The β_i spacing should to be determined so that a reasonably large acceptance rate is obtained for **each** pair. This can be done by a recursion, which is a slight modification of one by Kerler and Rehberg.

Let us denote by a_i^{pt} the **acceptance rate of the** $\beta_{i-1} \leftrightarrow \beta_i$ **exchange**. Assume $a_i^{pt} > 0$ for all $i = 1, \ldots, n-1$. Iteration m is performed with the β values

β_i^m, $i = 0, \ldots, n-1$. We define the β_i^{m+1} values of the next iteration by

$$\beta_0^{m+1} = \beta_0^m \quad \text{and} \quad \beta_i^{m+1} = \beta_{i-1}^{m+1} + a_i^m \left(\beta_i^m - \beta_{i-1}^m\right) \quad \text{for} \quad i = 1, \ldots, n-1 \ ,$$

where

$$a_i^m \ = \ \lambda^m \, a_i^{pt,m} \quad \text{with} \quad \lambda^m = \frac{\beta_{n-1}^m - \beta_0^m}{\sum_{i=1}^{n-1} a_i^{pt,m} \left(\beta_i^m - \beta_{i-1}^m\right)} \ .$$

For large acceptance rates a_i^{pt} the distance $\beta_i^m - \beta_{i-1}^m$ is increased, whereas it shrinks for small acceptance rates. The definition of the a_i coefficients guarantees $\beta_{n-1}^{m+1} = \beta_{n-1}^m$, *i.e.* the initial $\beta_{n-1} - \beta_0$ distance is kept.

A problem of the recursion is still that the statistical noise stays similar in each iteration step. It may be preferable to combine all the obtained estimates with suitable weight factors w^k. The final estimate after m iterations reads then

$$\beta_i = \frac{\sum_{k=1}^m w^k \beta_i^k}{\sum_{k=1}^m w^k} \ .$$

Two reasonable choices for the weights are

$$w^k = \min_i \{a_i^{pt,k}\} \ ,$$

which determines the weight from the worst exchange acceptance rate and suppresses the statistics of runs with a low w^k, and

$$w^k = \frac{1}{\sqrt{\sigma^2}} \quad \text{with} \quad \sigma^2 = \sum_{i=1}^{n} \left(\frac{1}{a_i^{pt,k}} \right)^2 \ ,$$

which relies on what would be the correct statistical weighting of the acceptance rates, if they were statistically independent (what they are not).

Computer implementation

We like to program the β exchange using MPI. Initially the indices of the β values are chosen to agree with the rank `MY_ID` of the process. The exchanges lead then to the situation for which the β indices become an arbitrary permutation of the numbers $0, \ldots, n-1$. Two approaches are possible:

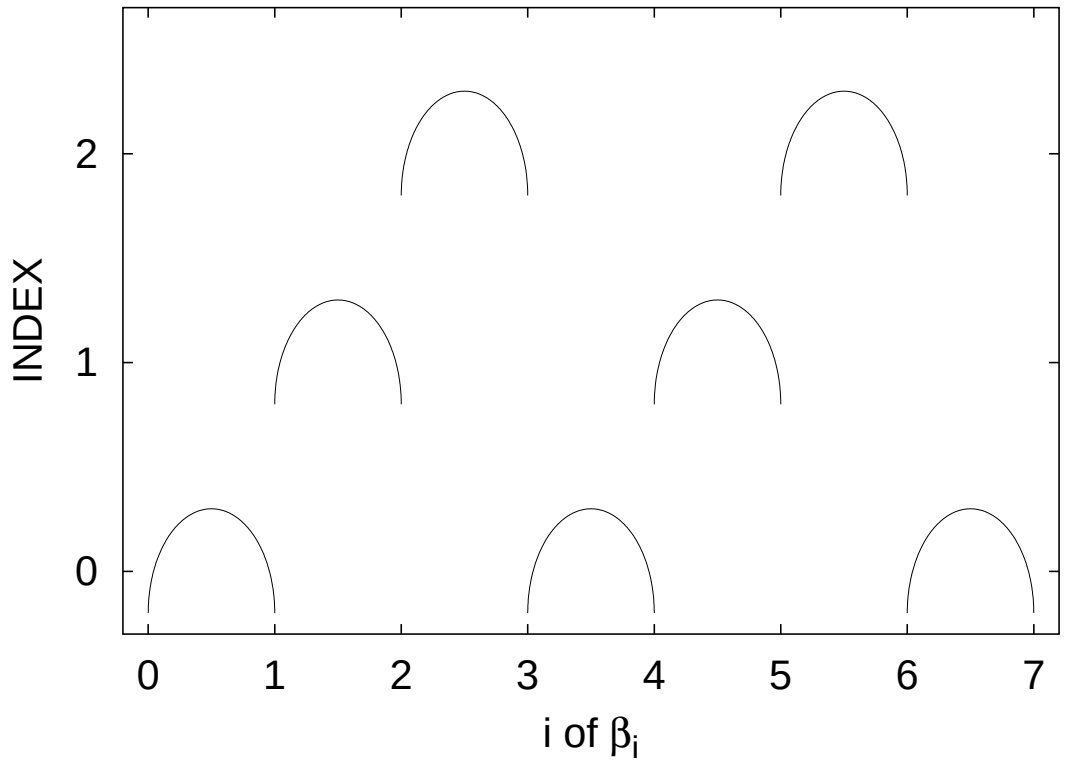
1. All the n processes send the needed information (the energies of their configurations) to a master processor. The master handles the exchange and sends each process its new β value back (which can agree with the old value).

2. The involved processes handle the exchanges through point to point communication.

We consider only the second approach, because for it the number of sweeps per time unit has a chance to scale linearly with the number of processors, whereas for

the first approach the master processor will become the bottleneck of the simulation when many processors are employed (when only few processors are used, it may be more efficient to let a master processor handle the exchanges).

Each process has to know the identities of the processes where the neighboring β values reside. They are stored in a neighbor array `NEIGH(2)`. The initial values are those defined in the `phbwrite_mpi.f` test program of `1MPI`. With each β exchange the nearest neighbor and next nearest neighbor connections get modified and a correct bookkeeping is the main difficulty of implementing the parallel tempering algorithm. We cannot deal with all pairs simultaneously, but need one buffer process between each pair. This leads to three distinct interaction groups, starting either with β_0, β_1 or β_2, which we label by `INDEX` $= 0, 1, 2$ as indicated for eight processes in the following figure ($e.g.$, for `INDEX` $= 0$ the buffer processes are $i = 2$ and $i = 5$):



Our MPI Fortran subroutine `p_pt_mpi.f` implements the β exchange (as in the figure the variable `INDEX` takes on the values 0, 1 or 2).

After the lowest β value is determined, the higher β values follow according to

the scheme of the figure. The thus identified processes send their `iact`, `IACPT_PT` (for a_i^{pt}) and `NEIGH(1)` values to their `NEIGH(2)` neighbors, *i.e.* to the processes which accommodate their upper, neighboring β values. These neighbor processes decide whether the β exchange is accepted or rejected, relying on the logical-valued function

$$\texttt{LPT\_EX\_IA}(\texttt{BETA1}, \texttt{BETA2}, \texttt{IACT1}, \texttt{IACT2}) \ ,$$

which implements the Metropolis criterion. For either outcome the processes send two integers back to their lower neighbors (the originally initiating processes) and one integer to their upper neighbors. If the exchange is rejected, the first element of each `MPI_SEND` is set to -2, thus signaling to the receiver that nothing changed. If accepted, the receivers store the addresses of their new neighbors in the `NEIGH` array. In addition, two more actions are performed: First, the initiating processes inform their left neighbors about the change by sending a new neighbor address to `NDEST3R`. This send is performed in any case, where the message -2 informs `NDEST3R` that nothing changed. Second, the processes whose β values changed re-calculate their heatbath weights and interchange also their parallel tempering acceptance rates `IACPT_PT`.

After a certain number of β exchanges, the β values are updated by either of the subroutines

`pt_rec0_mpi.f`, `pt_rec1_mpi.f` or `pt_rec2_mpi.f` .

In practice it may be unavoidable that one (or several) of the acceptance rates a_i^{pt} are zero. Applying the Kerler-Rehberg recursion blindly gives the undesirable result $\beta'_{i-1} = \beta_i$. To avoid this, we

$$\text{replace all } \; a_i^{pt} = 0 \; \text{ values by } \; a_{\min}^{pt} = 0.5/N_{\text{update}}$$

where N_{update} is the number of updates done. This works when there are enough β_i values to bridge the $[\beta_1, \beta_{n-1}]$ interval. Otherwise, the only solution to the problem of $a_i^{pt} = 0$ values is to increase the number of processes.

We could gather the needed permutation of β values, as well as their corresponding exchange acceptance rates, using point to point MPI communication.

However, it is shorter to rely on the

```
MPI_ALLGATHER (SENDBUF, ISENDCOUNT, SENDTYPE, RECVBUF,
               IRECVCOUNT, RECVTYPE, COMM, IERROR)
```

instruction. The arguments of `MPI_ALLGATHER` are: `SENDBUF`, the starting element of the send buffer; `ISENDCOUNT` (integer), the number of elements in the send buffer; `SENDTYPE`, the data type of the send buffer elements; `RECVBUF`, the first element of the receive buffer; `IRECVCOUNT` (integer), the number of elements in the receive buffer; `RECVTYPE`, the data type of the receive buffer; `COMM`, the communicator; `IERROR`, the MPI error code.

When a very large number of processes is involved, one may want to reprogram this in form of point to point interactions.

Finally a usefull instruction, in particular for debugging purposes, is

```
MPI_BCAST(BUF, ICOUNT, DATATYPE, ISOURCE, COMM, IERROR)
```

which broadcasts the buffer `BUF` from the process with rank `ISOURCE` to all processes specified by the communicator `COMM`. The number of elements in the buffer is given by `ICOUNT`, the datatype by `DATATYPE`, and `IERROR` is the MPI error code.