Explicit Parallelization of Robert-Bonamy Formalism

John Styers and Robert Gamache
University of Massachusetts Lowell

Abstract

The Robert-Bonamy formalism has long been employed for the computation of line shape parameters for atmospherically important molecules. As a method, it presents a fine balance between accuracy, and computational viability. While within the bounds of present-day computational resources, its calculations still constitute a significant amount of computational overhead. The vast majority of said computational demand is in the computing of the resonance functions. Major aspects of the calculation of the resonance function are extremely repetitive in nature−presenting a problem which is almost "embarassingly parallel." The computation of the resonance functions has been explicitly parallelilized resulting in an order of magnitude speed-up on local Macintosh machines−and multiple orders of magnitude speed-up on two Cray Supercomputers (Darter and MGHPCC). This will facilitate further scientific investigation.

Method

Equations and Approximations:

Starting from an unperturbed system, we introduce a perturbation.

This yields: (Anderson, 1949)

$$I(\omega) = const.\times\omega^4\Sigma_{abcdefg}(a|\rho_0|b)$$
$$\times\int_{-\infty}^{\infty} dt e^{i\omega t}(b|T^{-1}(t)|c)(c|\mu_0|d)\times\exp(i\omega_{ed}t)(d|T(t)|e)$$
$$\times\int_{-\infty}^{\infty} dt' e^{-i\omega t'}(e|T^{-1}(t')|f)(f|\mu_0|g)\times\exp(-i\omega_{fg}t')(g|T(t')|a)$$

In physics, one *FREQUENTLY* has to make assumptions/ simplifications. These–invariably–have to be justified.

1) Assume binary collisions

2) Full quantum mechanical modeling of interaction exceeds even present day computational resources for molecules of atmospheric interest

Substituting in for the various T's, one obtains:

Zeroth Order

$$S_0(b) = 0$$

First Order

$$S_1(b) = i\left[\sum_{m_i,m_2}\frac{(j_im_ij_2m_2|P|j_im_ij_2m_2)}{(2j_i+1)(2j_2+1)} - \sum_{m_f,m_2}\frac{(j_fm_fj_2m_2|P|j_fm_fj_2m_2)}{(2j_f+1)(2j_2+1)}\right]$$

Second Order

$$S_2(b)_{outer} = \frac{1}{2}\left[\sum_{m_i,m_2}\frac{(j_im_ij_2m_2|P^2|j_im_ij_2m_2)}{(2j_i+1)(2j_2+1)} + \sum_{m_f,m_2}\frac{(j_fm_fj_2m_2|P^2|j_fm_fj_2m_2)}{(2j_f+1)(2j_2+1)}\right]$$

$$s_2(b)_{middle} = -\sum_{m_i,m_i',m_f,m_f',m_2,m_2',M}\sum_{j_2'}\frac{(j_flm_fM|j_im_i)(j_flm_f'M|j_im_i')}{(2j_i+1)(2j_2+1)}$$
$$\times(j_fm_fj_2m_2|P|j_f'm_f'j_2'm_2')(j_i'm_i'j_2'm_2'|P|j_im_ij_2m_2)$$

$$S_2(b) = S_2(b)_{outer} + S_2(b)_{middle}$$

Semi-classical trajectories from solving Hamilton's Eqs.

Numerical Method:

Fully explicitly parallel, FORTRAN and MPI

Parallelization:

The code has been explicitly parallelized both for the efficiency of the parallelization and improved time performance of the code.

A bit about the Parallelization

The system domain is distributed on the various processors as a function of v and b (velocity and impact parameter).

Initially, an identical 2-dimensional array is built, on all processors:

```
do i = 1, FTD.NV
  do j = 1, FTD.ARR.NB
   work_tasks((i-1)
*FTD.ARR.NB + j,1) = i
   work_tasks((i-1)
*FTD.ARR.NB + j,2) = j
  end do
end do
```

The unique (and sequential) processor ID's run from 0 to n-1 (n == total number of processors)

Task is the variable that identifies the number of times one has cycled through the processors.

Therefore, task + procid yields which velocity and impact parameter one should be working on, i.e.

work_tasks((task + procid), 1) = curent velocity
work_tasks((task + procid), 2) = current impact parameter

This enables each processor to "know" what it should be working on.

At the end of the main loop, task is incremented by the number of processor, i.e.

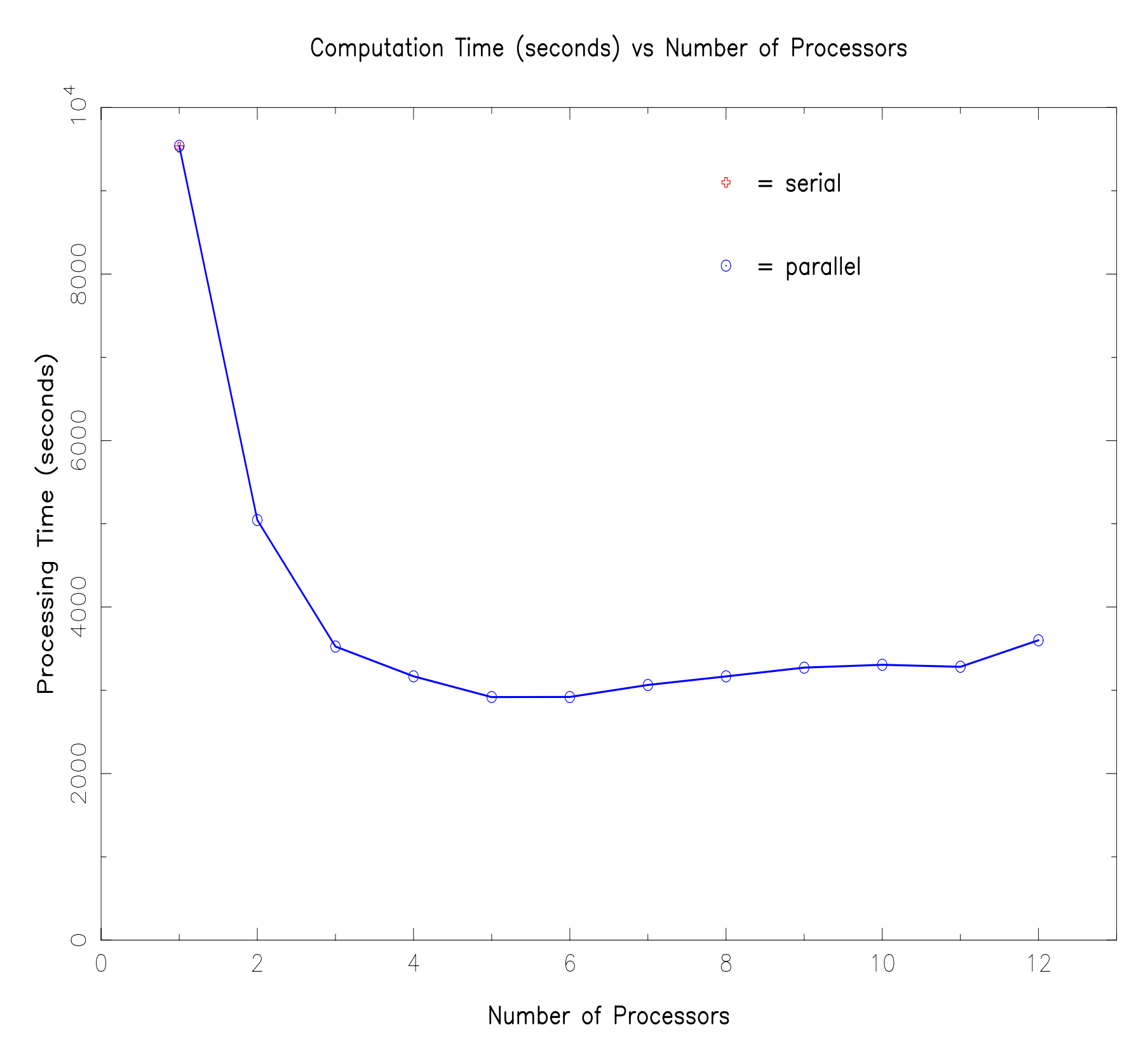
task = task + num_procs

Note that if task + num_procs is greater than the total number of tasks a "global" logical variable (work) keeps the code from performing "non-existent" work.

If an error is detected all processors must be called to terminate cleanly. If an error occurs on any processor a globally-scoped, logical variable (fault) is set to true.

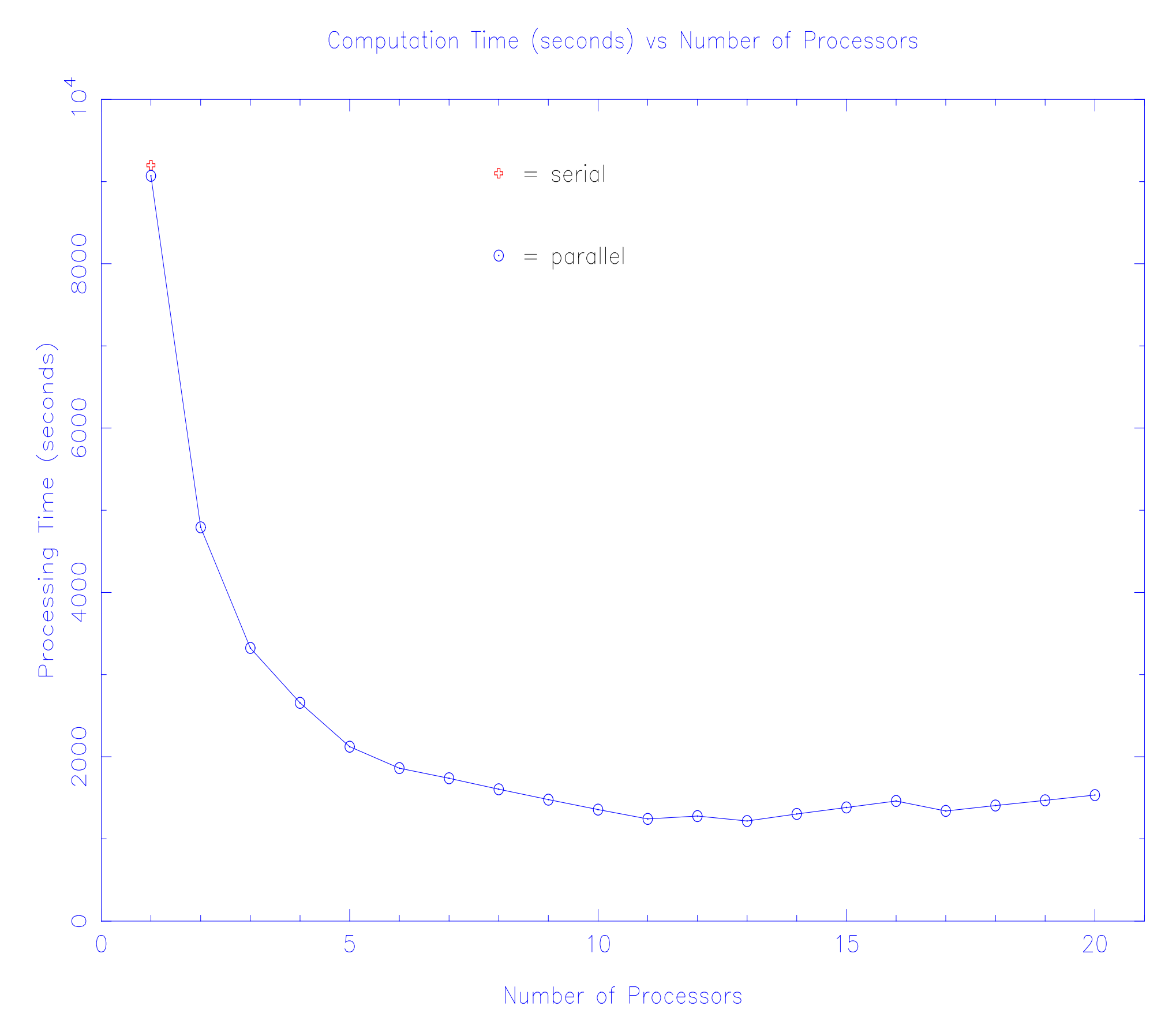
The code will periodically check to see if fault is true on any processor. If it is, fault is set to true on all processors and all processors are brought—cleanly—to a halt.

Speedup on Local Machines I



Performance plot for a four processor Mac(standard run)−showing an approximate four-fold speed-up.

Speedup on Local Machines II



Performance plot for a twelve processor Mac (standard run)−showing an approximate twelve-fold speed-up.

Unique Parallel Output Scheme

Consider a three-dimensional array:

a = 1,2,3, . . ., n
b = 1,2,3,4, . . ., n'
c = 1,2,3, . . ., n"

The progression is

111, 112, 113, 121, 122, 123, 131, 132, 133, 211, . . .

Now consider x(b,a,c) (Which is in essence the structure of the output of the code.). The progression is

111, 112, 113, 211, 212, 213, 311, 312, 313, 411, 412, 413, 121, 122, 123, 221, 222, 223, 321, 322, 323, 421, 422, 423, . . .

The problem presented is that the data must be outputted in this manner. This is complicated by this being a fully parallel (perhaps "massively parallel") system and by development for an arbitrary number of processors.

This required the development of a parallel output scheme

The code runs over the outer two limits whilst keeping the middle one fixed, i.e. one "b" (impact parameter) per processor (note, one can't just write out one "blob," after another).

So, a new subroutine was written that reproduces the write behavior of the serial code.

The code outputs its data into a new structure (FTD_MSTR), that then is dumped (when full) by the master processor (0), when "global_write" (a logical variable) is "true."

It then resets FTD_MSTR, by a call to (the subroutine) FTD_MSTR_CLR.

The code uses some mod statements to make certain that the next writes to FTD_MSTR, "land" in the correct places.

The procedure functions flawlessly and for an arbitrary number of processors, representing a significant accomplishment of code design and (fully parallel) implementation.

The Port from Hell

The porting of a code from one system to another is invariably a tedious and time-consuming affair. Dealing with the various compiler and linking issues can take as long as a week and half of solid work. The port from the local Mac to Darter has taken over 1 & ½ *months so far*.

Our problems included, but were not limited to:

- Multiple very serious linking problems
- A horrendous "final link" problem (which took the expertise of two professional computer scientists, Intel, and myself to resolve)
- A "deformed executable"
- Freeing up almost a Gig (!) of statically-linked memory
- A full-blown compiler error (only the second of my entire career)
- Further "linker confusion," during the final debugging

Summary and Conclusions

The parallelization of the code has been completed, resulting in an order of magnitude speed-up on local systems –and it is predicted will provide a speed-up of several orders of magnitude on the Darter and MGHPCC (Massachusetts Green High Performance Computing Center) supercomputers.

Significant obstacles were overcome (both in terms of code design and implementation), to achieve this goal. Generalized data output, for an arbitrary level of processors, proved particularly difficult. The porting of the code to the supercomputer Darter–as mentioned above –proved almost surrealistically difficult.

The code has fully validated. It results have been shown to be identical down to the byte level on the four processor Mac (This is for an over 100 Mbyte dataset.). It has validated down to the four or fifth decimal point, for half-width calculations on a twelve processor Mac. (It is believed that the (subtle) difference is due to the use of the "-O3" and "-parallel" optimizations during the half-width and line-shift calculations.)

Multiple orders of magnitude speed-up (with full validation of results) will greatly facilitate further scientific investigation.

Acknowledgments

This work is supported by NSF grant AGS-1156862

Computation is supported on Darter, NICS and the Massachusetts Green High Performance Computing Facility (MGHPCC),