Parallel integral transformation in computational quantum chemistry

Garold Murdachaew,[*] Nirmal Seenu,[e†] Omololu Akin-Ojo,[‡] and Zhao-Hui "Joe" Yang[§]

Advisor: Professor Krzysztof Szalewicz

Department of Physics and Astronomy and [e] *Computer and Information Sciences,*

University of Delaware

(December 9, 2000)

[*]garold@physics.udel.edu

[†]seenu@cis.udel.edu

[‡]prayerz@physics.udel.edu

[§]yangzh@udel.edu

Abstract

We parallelized the two-electron integral transformation to gain experience for running our quantum chemistry codes on a cluster with distributed memory and disk storage. The program whose results are presented here was parallelized through data distribution. The results show that the speedup for the full transformation is nearly linear for large enough problem sizes with a relative efficiency of over 90%. The largest problem size considered thus far, the Ar–HF van der Waals complex in a medium-sized basis set of $N^4 = 95^4$ basis functions, showed a relative speedup $T_5/T_{32} \approx 6$. Extrapolating the nearly linear curve to the time on one processor yields a relative speedup and efficiency on 32 processors of about 30 and 94%, respectively. Remaining sequential bottlenecks, scalability to larger problem sizes, and proposed improvements are addressed in detail below.

I. INTRODUCTION

Computational quantum chemistry can now obtain fairly accurate results for small molecules and clusters. However, calculations of even modest accuracy for large systems (e.g., biological and pharmaceutical molecules and clusters) may still be intractable due to the speed, memory, and storage limitations of present-day computers. The advent of cheaper and more accessible supercomputing power through the development of clusters with multiple nodes connected by a fast network calls for continued advances in the parallelization of quantum chemistry codes (and for the optimization of the underlying serial algorithms) to allow medium-sized problems to be treated faster and with an increased accuracy and also to allow some larger problems to be addressed for the first time. Since its memory/storage requirements and computation time vary with problem size as $\mathcal{O}(N^4)$ and $\mathcal{O}(N^5)$, respectively, a significant bottleneck in such calculations and a good candidate for algorithmic optimization and parallelization is the two-electron integral transformation.

To more fully see the need for parallelization, at least in the context of speeding up calculations, a concrete example should be presented. Consider a topic that remains of interest, water. Chemists know that water derives its properties from the collective action of hydrogen bonds. To obtain a more quantitative understanding, quantum chemists calculate and fit points on the potential energy surface [1] defining the interaction between the simplest form of water, the water dimer. A recent such first-principles potential surface[1] required the calculation of 2510 points. Each of these points took about 1 hour to calculate on a single processor of an SGI Origin 2000 and the various transformations involved in the calculation took about half of this time. The basis set used was of medium-size, $N^4 = 112^4$ basis functions. In fact, this calculation was trivially parallelized by running different points on different processors. The issue of parallelization of transformation (and of many other steps

[1]Interaction energy as a function of the dimer's 6-dimensional geometric configuration (each water molecule is approximated as being rigid).

in a quantum chemical calculation) becomes more pressing when one considers large problem sizes. Current research grade problems may involve much larger basis sets (e.g., $N^4 \approx 400^4$ basis functions). Such a problem becomes impossible to run on most single processors due to its huge disk and memory requirements. Of course, if it does run at all, the time for the calculation to complete may be too large to be practical.

II. OVERVIEW OF COMPUTATIONAL QUANTUM CHEMISTRY

A brief overview of quantum chemistry and its computational aspects follows. For more details the reader is referred to the standard texts.[2,3] Computational quantum chemists (or chemical physicists) model many-electron atoms, molecules, or clusters and make predictions of their properties. Examples of such properties may be the equilibrium and transition state structures of the molecules or clusters, and their ground and excited electronic states. In the Born-Oppenheimer approximation which separates the motion of light electrons from those of heavy nuclei, the determination of all such properties requires first of all the solution of the electronic Schrödinger equation for many sets of fixed nuclear positions. (This gives points on the potential energy surface of the system.) Many-electron systems do not allow an analytic solution and thus a numerical method must be used. The molecular wavefunction is written as a finite expansion with adjustable coefficients in the single electron spin orbitals of the atoms comprising the molecule or cluster. The problem would neatly separate into individual problems for each electron but for the terms such as $1/r_{12}$ describing the repulsion between any two electrons. As a zeroth order approximation, such terms are first treated in an average way allowing one to obtain a self-consistent solution. This approach is known as the Hartree-Fock (HF) method. Since it treats the instantaneous electron-electron repulsion in a self-consistent or average way, it is also known as the self-consistent field (SCF) method.

The HF problem may be solved iteratively to obtain the molecular energy and corresponding molecular wavefunction (i.e, the values of the expansion coefficients for the one-electron orbitals) at the HF level. However, this zeroth order approximation to the solution in many applications does not give results of sufficient accuracy. What is required is a higher "dynamic correlation" treatment of the electron-electron interaction which properly correlates the instantaneous repulsion between any two electrons. Correlation corrections can be obtained in a number of ways. For example, perturbation theory can be used to correct the molecular HF energy and wavefunction. An important intermediate step in obtaining any correlation corrections past the HF level is the "integral transformation."

In the course of obtaining the HF solution, two basic types of atomic integrals are calculated. These are the integrals involving a product of a one-electron operator between two electronic orbitals (one-electron integrals) and the integrals involving a product of a two-electron operator between four electronic orbitals (two-electron integrals). The former integrals can be identified by two indices (thus forming a matrix or a tensor of rank 2) and the latter by four indices (thus forming a tensor of rank 4). However, the molecular energy expressions are usually written in terms of integrals over the molecular orbitals while the HF method obtains integrals over the atomic orbitals.

At this stage it may be useful to re-examine the starting approach if the object of the calculation is to obtain correlation corrections to the HF energy. The traditional approach, as suggested above, calls for first solving the HF problem and then transforming the atomic integrals to molecular integrals, thus allowing the correlation corrections to be readily calculated in a third step. At the start of the transformation the atomic integrals are read into memory from disk. This I/O may be very time-consuming for large problems (e.g., in the current research grade problems being considered, in the traditional approach more than 20 GB of data may need to be read into memory at the start of the transformation).

The new integral-direct approach solves the potential I/O problem by first recasting the correlation-energy corrections most easily written in terms of molecular integrals into the corresponding more complicated expressions in terms of the atomic integrals. Then, as the correlation corrections are requested, the required atomic integrals can be calculated on the fly. Thus the need for extensive I/O is eliminated. Another reason for considering the direct approach is technological: advances in processor speed and memory are far outpacing advances in storage. However, in this work we assume that the transformation routine is part of a package built on the traditional approach and is thus called with the atomic integral list on disk storage as input data. The reader is referred to the references for more information on the parallelization of transformation in the context of the integral-direct approach.[4–8]

Large problems will also have large memory requirements. In either approach, this need can be met by utilizing the aggregate memory of a cluster (after parallelization, of course).

III. TWO-ELECTRON INTEGRAL TRANSFORMATION AND ITS PARALLELIZATION

Traditional quantum chemistry calculations with correlation corrections require that two basic types of transformation be performed. The one-electron transformation is relatively straightforward and not so storage-hungry or time-consuming. It will not be addressed here. The two-electron transformation is the one that presents difficulty in practice due to its storage and time demands, as described above. In terms of one-electron orbitals, the atomic two-electron integrals are written as

$$(\mu\nu|\lambda\sigma) = \int d^3\mathbf{r}_1\, d^3\mathbf{r}_2\; \phi^*_\mu(1)\phi_\nu(1) r_{ij}^{-1} \phi^*_\lambda(2)\phi_\sigma(2). \tag{1}$$

Thus the two-electron integrals form a tensor of rank 4. From its definition, the tensor has eight symmetries,

$$(pq|rs) = (qp|rs) = (pq|sr) = (qp|sr) = (sr|qp) = (rs|qp) = (sr|pq) = (rs|pq). \tag{2}$$

Thus the required storage space is reduced to $N^4/8$. [2] The job of the two-electron transformation is to successively multiply different portions of this tensor by a two-dimensional matrix holding the transformation coefficients between the atomic and the molecular orbitals.

---

[2] In practice, a threshold is also defined such that integrals below the threshold are not stored. Zero values are not stored either. All of this is helpful in reducing the required storage space.

The equation defining the transformation involves four such matrix-tensor multiplications and can be written as

$$(ij|kl) = \sum_{pqrs} C_{pi} C_{qj} C_{rk} C_{sl} (pq|rs). \tag{3}$$

Since this equation has a time complexity of $\mathcal{O}(N^8)$, applying it directly is not efficient. A more efficient approach is the so-called N5 algorithm,

$$\begin{aligned} I_{pjrs} &= \sum_s C_{qj} I_{pqrs}, \\ I_{ijrs} &= \sum_r C_{pi} I_{pjrs}, \\ I_{ijrl} &= \sum_q C_{sl} I_{ijrs}, \\ I_{ijkl} &= \sum_p C_{rk} I_{ijrl}. \end{aligned} \tag{4}$$

(Here, to emphasize the tensor nature of these objects, we have changed notation so that $(pq|rs) = I_{pqrs}$, etc.). The N5 algorithm proposes that the transformation be performed in four steps, each with time complexity of $\mathcal{O}(N^5)$, a significant improvement.

With an eye to parallelization by data distribution of I_{pqrs}, we see that its two pairs of adjacent indices can be conveniently transformed in two separate stages:

$$I_{ijrs} = \sum_p \sum_q C_{pj} C_{qj} I_{pqrs} = \sum_p C_{pj} [\sum_q I_{rspq} C_{qj}] \tag{5}$$

and

$$I_{ijkl} = \sum_r \sum_s C_{rk} C_{sl} I_{ijrs} = \sum_r C_{rk} [\sum_s I_{ijrs} C_{sl}]. \tag{6}$$

This suggests that we rewrite these operations as simple 2-dimensional matrix operations, one such operation per stage for each set of the index pair unaffected in that stage. These stages (subroutines, actually) are called `tr1` and `tr2` and can be written as

$$\texttt{tr1}\text{: for each } rs \text{ pair, } Irs' = C^{\mathrm{T}}[Irs''C] \tag{7}$$

and

$$\texttt{tr2}\text{: for each } ij \text{ pair, } Iij = C^{\mathrm{T}}[Iij'C]. \tag{8}$$

The 2-dimensional matrix expressions allow us to use the efficient FORTRAN level 3 `blas` (Basic Linear Algebra Subroutines[9]) library routines `DSYMM` and `DGEMM` for the matrix multiplications in each stage. Using `DSYMM` takes advantage of the $p \leftrightarrow q$ and $r \leftrightarrow s$ symmetries, a quarter of the symmetry of I_{pqrs} and reduces the operation count in that half of each of the two stages by a factor of about 1/2. In each stage, after the first multiplication using DSYMM this symmetry is destroyed and we use `DGEMM` which however allows us to efficiently do the initial required transpose of the C matrix.

Now we describe the data distribution in the parallel code. In Eqs. (7) and (8), the Ixy ($xy = rs, ij$) are all temporary 2-dimensional matrices, one such matrix for each set of the identifier pair xy. In the parallelization, prior to `tr1` the original tensor has been broken up into N^2 N^2-sized matrices Irs'', one for each rs pair, with each processor operating on N^2/P such matrices, where P is the number of processors. [3] Thus, after the end of `tr1`, each processor will hold N^2/P of the half-transformed matrices Irs'. These matrices can be stored in such way as to allow their optimum redistribution so that at its end each processor will hold N^2/P of the re-distributed half-transformed matrices Iij' necessary for `tr2`. Properly programmed, this redistribution becomes a transpose over all processors of the half-transformed tensor I_{ijrs}. The goal of the transpose is to take the tensor left scattered by rs (each processor originally holding all pq and then all ij) at the end of `tr1` and re-scatter it by the ij (each processor holding all rs) in preparation for `tr2`. The subroutine performing this re-distribution is called `comm2`. After `tr2` has completed, the fully transformed matrices Iij remain on the processors distributed by ij and ready for collection or any additional calculations.

Let us further describe the communication subroutines, `comm1` and `comm2`. In `comm1`, the

---

[3]The data distribution and consequent load balancing is designed to be robust: the maximum difference in load between any two processors is one xy pair since a non-zero remainder of N^2/P is dealt out to the processors.

packed indices and integrals are broadcast to all processors (this is somewhat inefficient—it has sublinear scaling—and would need to be improved in a future version of the code), while in `comm2` `MPI_ISEND` and `MPI_IRECV` are used to transpose the half-transformed tensor across all the processors. This is an efficient operation since the number of messages is the minimum possible and message lengths are maximized. This is doable since the preceding step—`tr1`—stored the data contiguously in the best possible way for this communication step.

The actual timing experiments and their analysis is described in detail below. Briefly, the timing experiments were performed as follows. Prior to running our program `trans_par`, we converted the data set obtained from the previous stage (the HF calculation) into the most suitable format. This required packing the indices (thus allowing us to store 4 4-byte integers—converted to 4 1-byte integers—as 1 packed 4-byte integer) and writing the atomic integral data as an unformatted file. [4] The combination of using a packed index and an unformatted file typically allowed us to reduce the size of the integral file to be stored prior to the start of the program—and to be read into the program—by a factor of about 1/3 (for example, for the $N^4 = 95^4$ problem size, the atomic integral file size shrank from 54 MB to 38 MB). The correctness of the program was verified for some small problem sizes. When getting our timings, we also confirmed that the determination of data distribution was robust and that the load was well-balanced for any combination of processors P and any of the problem sizes considered.

[4] These techniques are often used in quantum chemistry calculations.

IV. ANALYSIS OF THE TIMINGS

A. subroutine `readdata`

The data size is of order N^4 but because of symmetry we need to store only 1/8 of the data. In addition, atomic integrals with values less than a certain threshold are not stored so that a further reduction by a factor $f \leq 1$ [5] comes in. Only the master processor reads the data hence the subroutine, which is essentially sequential goes as

$$T_{\texttt{readdata}} \sim \mathcal{O}(fN^4/8).$$

This is independent of the number of processors as can been seen from Figs. 1-3. (Note that all times given here and below are essentially wall times.)

B. subroutine `comm1`

In this subroutine, the master processor, having read the data, broadcasts the data to all the other processors. The amount of data to be broadcast is $\mathcal{O}(fN^4/8)$ and, since the broadcast is done via a *binary tree*, the wall time for this subroutine is

$$\begin{aligned} T_{\texttt{comm1}} &\doteq (\frac{fN^4}{8}4t_w + 2t_s)2\log_2 P \\ &+ (\frac{fN^4}{8}8t_w + 2t_s)2\log_2 P \\ &= (\frac{fN^4}{8}12t_w + 4t_s)2\log_2 P. \end{aligned}$$

Here, t_w is the time required to send 1 byte (and in this case, we send 4-byte integers—for the packed indices of the integrals—and 8-byte real numbers for the integrals themselves); t_s is the startup time or latency time for a communication; and P is the number of processors. This process is not exactly sequential but increases *slowly* with P (because of the $\log_2 P$

[5] Typically, $f \approx 0.4$.

factor). This can be seen from the speedups (Fig. 3c, for example).

C. subroutine `tr1`

In here, three major things are done. First, the indices are unpacked (this takes a time of $\mathcal{O}(N^4)$ since we have this much data to deal with), and integrals belonging to that processor (including symmetry identical integrals and zero-value integrals) are stored on that processor's local data structure. Secondly, the actual matrix multiplications are performed. Since there are $\mathcal{O}(N^5)$ floating point operations to be done but this is shared essentially equally by all the processors, this transformation takes a total time of $\mathcal{O}(N^5/P)$ and this is exactly parallel. The third step is to re-pack the indices and this essentially takes a time $\mathcal{O}(N^4)$. Hence the total time taken for `tr1` is

$$T_{\texttt{tr1}} \sim \mathcal{O}(N^4) + t_{\text{comp}}\frac{N^5}{P}$$

, where t_{comp} is the time for one floating point operation.

D. subroutine `comm2`

This is the second communication in which each processor sends a piece of its data to each of the other processors. Now, each processor has a Pth of the total data (i.e., N^4/P). It sends a Pth of this to each of the other $P-1$ processors hence the wall time for this process is

$$\begin{aligned} T_{\texttt{comm2}} &\doteq 4t_w 2(P-1)\frac{1}{P}\frac{N^4}{P} + 2t_s(P-1) \\ &+ 8t_w 2(P-1)\frac{1}{P}\frac{N^4}{P} + 2t_s(P-1), \end{aligned}$$

so that

$$T_{\texttt{comm2}} \rightarrow 24t_w(\frac{N^4}{P}) + 4t_s(P-1)$$

. Hence, if N is small (say 10) and t_s is larger than t_w by a factor of more than about $10^3 - 10^4$, then the wall time for communication increases as the number of processor increases (i.e., the second term, $4t_s(P-1)$, becomes significant or even dominates) (see Fig. 2a) but if N is large enough (say 95 and above) then the first term dominates so that the wall time for communication (`comm2`) falls as the number of processor increases. (See Fig. 2c.)

E. subroutine `tr2`

This is essentially like `tr1` (except that we do not re-pack the indices—but this has a negligible effect as can be seen from Figs. 2a-c). Hence, the time taken is similarly

$$T_{\texttt{tr2}} \sim \mathcal{O}(N^4) + t_{\text{comp}} \frac{N^5}{P}$$

.

F. program `trans_par`

The total time taken for the whole program `trans_par` is the sum of the times for the all the subroutines, i.e.,

$$\begin{aligned} T_P \sim\ & T_{\texttt{readdata}} + T_{\texttt{comm1}} \\ & + T_{\texttt{tr1}} + T_{\texttt{comm2}} + T_{\texttt{tr2}} \\ & \doteq \mathcal{O}(fN^4/8) + (\frac{fN^4}{8} 12t_w + 4t_s) 2\log_2 P + \mathcal{O}(N^4) \\ & + t_{\text{comp}} \frac{N^5}{P} + 24t_w(\frac{N^4}{P}) + 4t_s(P-1) \\ & + \mathcal{O}(N^4) + t_{\text{comp}} \frac{N^5}{P} \end{aligned}$$

Clearly, the terms in N^5 dominate and since this term is essentially parallel (i.e., it goes more specifically as N^5/P), we observe the behavior shown in Figs. 1a and 2a-c, i.e., the total times go as $1/P$.

V. CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

A. Conclusions

One important thing that stands out in this work is that it is worthwhile to parallelize the integral transformations program. By this we mean that parallelizing this program leads to benefits. Two main advantages of parallelizing this project are :

1. The high speedup obtained which means that with this parallel version, the time taken to do a particular transformation falls as the number of processors increases.

2. Bigger jobs are (now) tractable. Jobs that could not be run on a single processor (because of the limited RAM size) using the sequential algorithm can now be run since the data is spread out over all the processors so that no one processor holds all the data any longer.

B. Suggestions for further work

Further work could take a look at the following:

1. The time for `readdata` could be reduced by splitting up the initial file on the disk into as many processors as there are so that all the processors could read the data all at once rather than waiting for the master processor to read in the data.

2. An *incore* and *outcore* algorithm could be written. In this program we have written, if the data size is so large that spreading out the data over the processors does not help (i.e., the amount of data for each processor exceeds its RAM size), then it would not be possible to run the program for that particular job without using more processors. This does not have to be the case. Further work could include the *incore* and now also *outcore* algorithms so that each processor writes its "overflow" data unto disk and reads from disk when it needs the data. In this way, one can still make do with

the available number of processors. (In fact doing this will give a superlinear speedup because the *incore-outcore* program on one processor should be really slow.)

3. In a real life application, a full transformation transformations is rarely needed and rarely done. Only $\mathcal{O}(N^4)$ are done so that the total wall time would no longer be dominated by an $\mathcal{O}(N^5)$ term in which case the speedups would not be as high we have observed here since now the other terms (for example the communication time) would no longer be so insignificant. Thus other ways to reduce the communication time would need to be devised.

4. The program could be run on different clusters with different architectures to see how the speedups and efficiencies behave.

5. Larger problem sizes ($N^4 > 95^4$) could be considered with more processors ($P > 32$).

VI. ACKNOWLEDGMENTS

We are grateful to Dr. R. Bukowski and A. J. Misquitta for many helpful discussions. GM wishes to acknowledge additional discussions with Prof. K. Szalewicz. We are also grateful to the instructors of the PHYS 838/CISC 879. (Profs. W. H. Matthaeus and L. Pollock) for the instruction in MPI.

REFERENCES

[1] E. M. Mas, K. Szalewicz, R. Bukowski, and B. Jeziorski, J. Chem. Phys. **107**, 4207 (1997).

[2] W. J. Hehre, L. Radom, P. v.R Schleyer, and J. A. Pople, *Ab initio Molecular Orbital Theory* (Wiley, New York, 1986).

[3] A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry* (Dover, Mineola, NY, 1996).

[4] M. Schütz and R. Lindh, Theor. Chim. Acta. **95**, 13 (1997).

[5] T. L. Windus, M. W. Schmidt, and M. S. Gordon, in *Parallel Computing in Computational Chemistry*, Vol. 592 of *ACS Symposium*, edited by T. G. Mattson (American Chemical Society, Washington, DC, 1995), p. 16.

[6] T. L. Windus, M. W. Schmidt, and M. S. Gordon, Theor. Chim. Acta. **89**, 77 (1994).

[7] T. L. Windus, M. W. Schmidt, and M. S. Gordon, Chem. Phys. Lett. **216**, 375 (1993).

[8] A. M. Márquez and M. Dupuis, J. Comp. Chem. **16**, 395 (1995).

[9] Basic Linear Algebra Subroutines, `http://www.netlib.org/blas`.

FIGURES

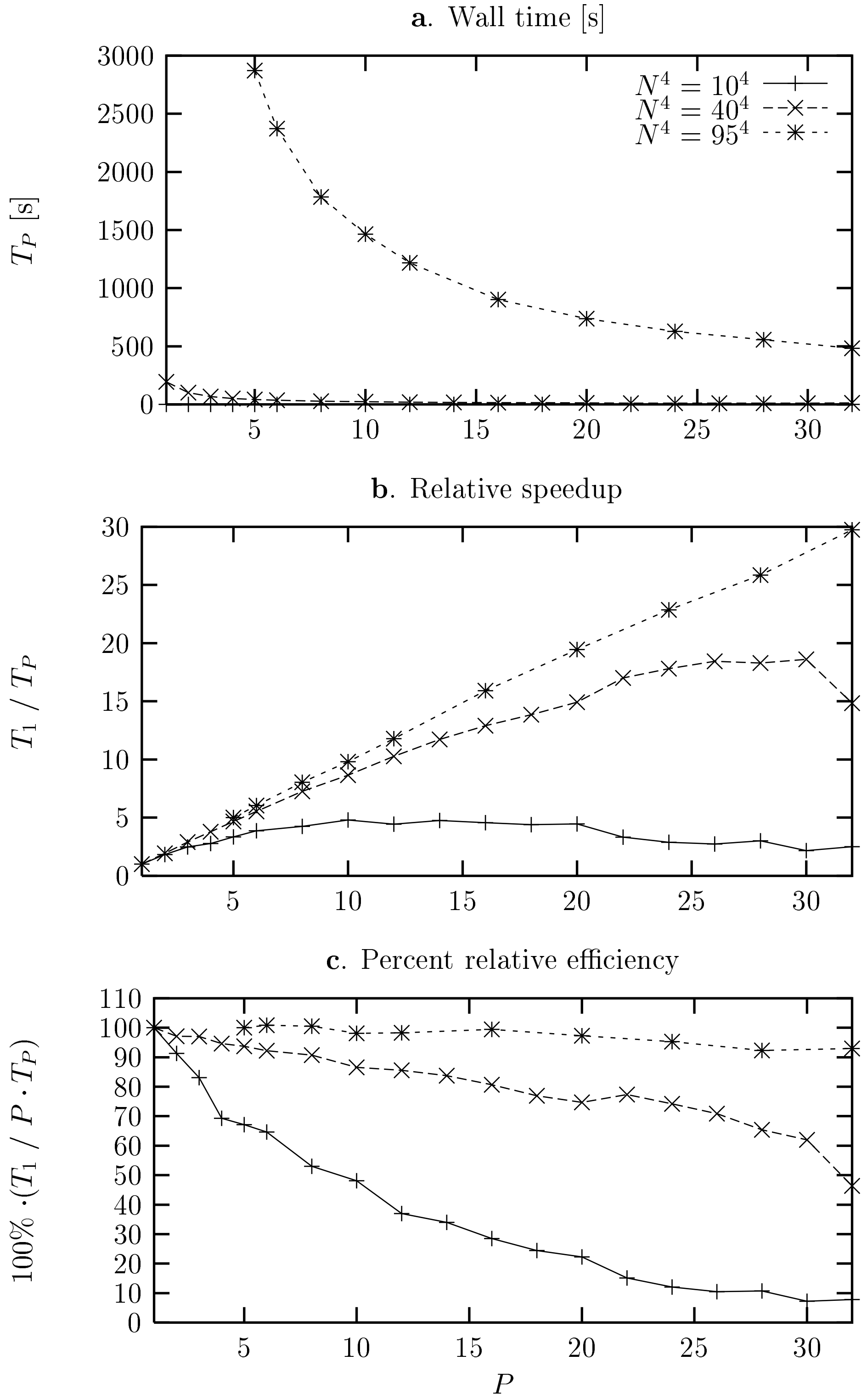

FIG. 1. Wall time (maximum of times over all processors), speedup, and efficiency of the whole program for various problem sizes and various number of processors.

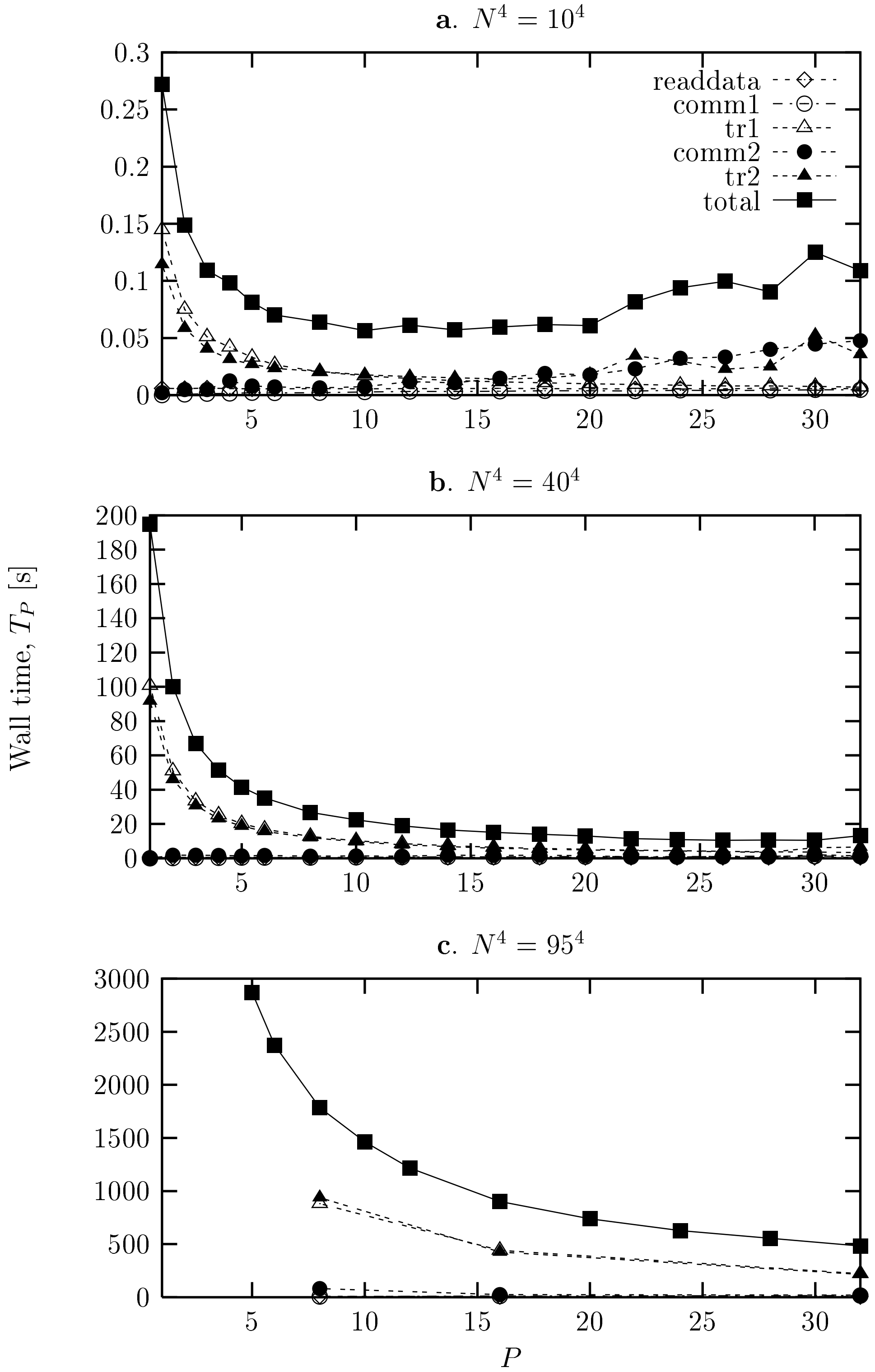

FIG. 2. Wall times of portions of the program (averages of portion times over all processors) and wall time of the whole program (maximum of times over all processors) for various problem sizes and various number of processors.

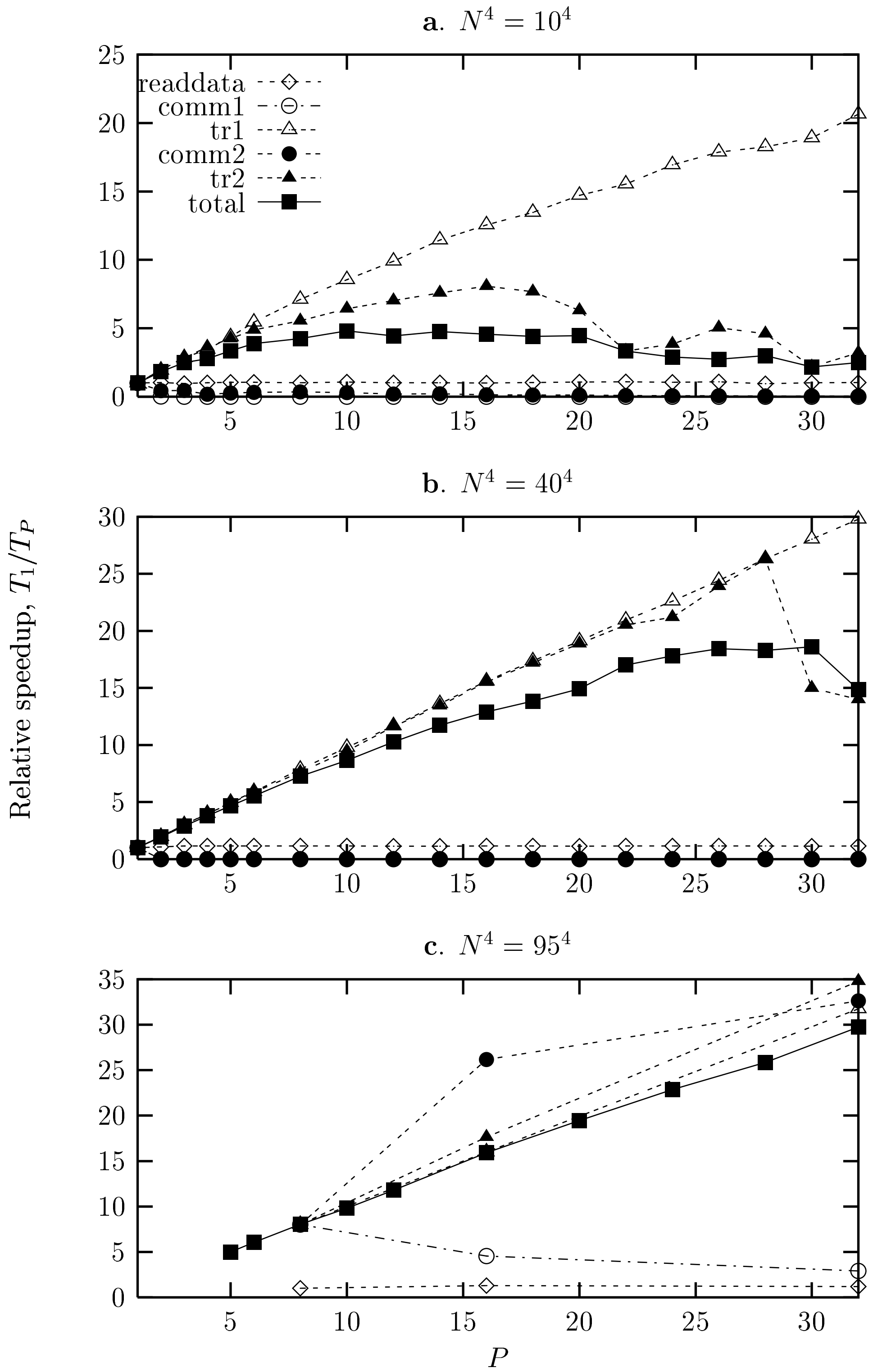

FIG. 3. Relative speedups of portions of the program and of the whole program for various problem sizes and various number of processors. (Note: The speedup for $N^4 = 95^4$ was extrapolated by extrapolating to the time on one processor.)