Yale University
Department of Computer Science

# Solving Schrödinger's equation on the Intel iPSC by the Alternating Direction Method

Faisal Saied, Ching-Tien Ho,
S. Lennart Johnsson, Martin H. Schultz

YALEU/DCS/TR-502
January 1987

This work was supported in part by the following: ONR grant N00014-84-K-0043, ONR grant N00014-86-K-0564, ONR grant N00014-82K-0184, NSF/CER grant DCR 8106181 and ARO FSP DAALO3-86-G-0029. Approved for public release: distribution is unlimited.

† To appear in Proceedings of the Second Hypercube Conference 1986.

# Solving Schrödinger's equation on the Intel iPSC by the Alternating Direction Method

Faisal Saied, Ching-Tien Ho,
S. Lennart Johnsson, Martin H. Schultz
Department of Computer Science
Yale University
New Haven, CT 06520

January, 1987

**Abstract**

We consider the numerical solution of the Schrödinger's equation and investigate several different algorithms for implementing the Alternating Direction Method on hypercubes. We indicate the relative merits of the algorithms depending on cube parameters such as arithmetic speed, communication latency, transfer rate, the packet size, and the cost of reordering data locally. We present timings for the Intel iPSC that show that Alternating Direction Methods can be implemented efficiently on hypercubes.

## 1 Introduction

In this paper, we discuss implementations of the Alternating Direction Method on the Intel iPSC. The implementations are for the time-dependent, two-dimensional Schrödinger equation on a rectangular domain. Schrödinger's equation is a fundamental equation in quantum mechanics. We focus on the efficiency of the Alternating Direction Method. The results reported here were obtained on a 32 node configuration of the Intel iPSC.

The structure of this paper is as follows. We describe the problem in which we are interested, and mention several numerical techniques for its solution. In this paper we investigate only the Alternating Direction Method. We briefly discuss the architectural features of the Intel iPSC that are relevant for our implementations. We outline several methods for solving multiple tridiagonal systems on a hypercube, indicate their relative performances on the iPSC and compare them on the basis of model times. We describe the basic approach towards the parallel implementation of the ADM that we have followed, and discuss several variants. Finally we present the results of some experiments on the iPSC.

## 2 The Model Problem

We consider a time-dependent partial differential equation of the following form:

$$i\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + V(x,y,t)u \text{ in a rectangular region, } R, \tag{1}$$

where $u = u(x,y,t)$ satisfies

$$u(x,y,0) = \phi(x,y) \text{ for } (x,y) \in R, \text{ and } u(x,y,t) = \psi(x,y) \text{ for } (x,y) \in \partial R.$$

This equation is essentially the time-dependent, two-dimensional Schrödinger's equation in quantum mechanics.

## 3 Numerical Schemes for the Model Problem

Some of the methods that have been applied to (1) are the following.

- Explicit methods
- Crank-Nicolson
- ODE techniques (method of lines)
- ADM (alternating direction methods)

An explicit scheme that is conditionally stable with first order accuracy in time was proposed in [1] for Schrödinger's equation. With two space dimensions, the computational stencil involves eight neighboring grid points. Each time step involves a matrix-vector product only, with local communication involving the stencil neighbors only. However, a larger number of time steps is required to achieve a given accuracy at a fixed point in time than with methods that are second order in time. Thus, an efficient parallel implementation of this method is probably not competitive with the ADM.

The Crank-Nicolson scheme is unconditionally stable, second order accurate in time and implicit. It requires the solution at each time step of a linear system whose matrix has the non-zero structure of the discrete Laplacian. Solving such a system can be costly, especially if the form of the function $V$ precludes the use of fast Helmholtz solvers. The efficient parallel implementation of iterative solvers of this class is a topic of current research, but is not considered in this paper.

In the ODE approach, one discretizes the space variables and solves the resulting coupled system of ordinary differential equations, which are typically stiff, by some implicit method, which again requires solving large sparse linear systems, defined by the stencils, at each time step.

Alternating Direction Methods were developed for solving parabolic partial differential equations [7]. They have the advantage of being unconditionally stable, second order accurate in time and require $O(P^2)$ operations per time step on a $P \times P$ grid. In applying the ADM to our model problem the spatial operators are approximated by their discrete equivalents, which for 3-point central differences yields an equation of the form

$$\frac{du}{dt} = \tilde{A}_x u + \tilde{B}_y u + f(x,y,t)u, \tag{2}$$

where $\tilde{A}_x$ and $\tilde{B}_y$ with the appropriate orderings, are block diagonal matrices, each block being a tridiagonal matrix. One ADM step for this equation consists of two half steps,

$$(I - \frac{1}{2}\Delta t A_x)u^{i+\frac{1}{2}} = (I + \frac{1}{2}\Delta t B_y)u^i,$$

$$(I - \frac{1}{2}\Delta t B_y)u^{i+1} = (I + \frac{1}{2}\Delta t A_x)u^{i+\frac{1}{2}}.$$

Here, $A_x$ and $B_y$ are obtained from $\tilde{A}_x$ and $\tilde{B}_y$ by modifying the diagonal entries to account for the last term in (2). In the first half step, we form $P$ tridiagonal matrix-vector products in the $y$-direction, corresponding to $(I + \frac{1}{2}\Delta t B_y)u^i$, to get the right hand sides for the $P$ tridiagonal solves in the $x$-direction which involve $(I - \frac{1}{2}\Delta t A_x)$. The second half step requires the same operations to be performed, with the roles of the $x$- and $y$-directions interchanged. It is usual to transpose the data at each half step in sequential implementations, to make the tridiagonal solves in both directions equally efficient.

For the applications we are interested in, the $P \times P$ grid vector $u$ is complex. We give the operation counts in terms of real, floating-point operations, even though complex arithmetic is used. Thus, a complex add is counted as two flops, etc. Let $C_{MVP}$ and $C_{TRID}$ be the number of real single precision operations required per unknown for each matrix-vector product and tridiagonal solve respectively. Further, let $t_a$ be the time to perform one floating-point operation. Then the sequential time for one ADM step on a $P \times P$ grid is

$$T(P) = 2(C_{MVP} + C_{TRID})P^2 t_a.$$

## 4 Solving Tridiagonal Linear Systems on Hypercubes

In this section, we discuss the problem of solving multiple independent tridiagonal systems on a hypercube. We first give a brief description of the methods we have considered and then compare them, in terms of actual performance on the iPSC and on the basis of model times.

The choice of the method for solving the tridiagonal systems is influenced by the size of the cube and the size of the local memories relative to the problem size, in addition to depending on other cube parameters like the arithmetic rate, the communication latency, the transfer rate, etc.

### 4.1 Outline of the Methods

Suppose that we have $P$ tridiagonal systems, each of order $Q$, and a hypercube with $N$ processors. For simplicity, we assume that $P$ and $Q$ are powers of 2. We will initially outline the methods for the case $N \leq P$ and using "one-dimensional domain decomposition". We will indicate the modifications that are necessary to implement them when more processors are available ($P < N$), using two-dimensional domain decomposition. Two-dimensional domain decomposition can also be applied when $N \leq P$, and will be discussed in the next section.

The simplest approach to the problem is to move $P/N$ systems to each processor, solve them locally, using standard Gaussian elimination and move the solutions back to their original distribution. Since the data movement is equivalent to a transpose (of a distributed, rectangular matrix), we will refer to this method as **TGET** [5] ("Transpose-GE-Transpose"). For Gaussian elimination, the number of floating point operations per row required is 8.

The transpose operation can be implemented efficiently as follows. Assume that there are $P$ systems each of order $Q$ spread identically across $N$ processors, each processor having $Q/N$ rows of each system. At the first step, each processor whose node number has its lowest order bit equal to zero, sends its rows of systems $(P/2) + 1, \ldots, P$ to its neighbor whose node number differs in the lowest order bit. That neighbor sends its rows of systems $1, \ldots, P/2$ in exchange. All processors then reorder the unsent part of their original data and the received data so as to make the data for each system contiguous and ordered. This process is repeated for the other bits in the node numbers, going from low to high. At each step, the number of systems in each processor is halved and the number of rows of each system in a processor is doubled. Thus in each of $\log N$ steps, each processor sends and receives messages equal to half its local data, and reorders all of its local data. Let $T_{comm}(B)$ be the time to send or receive a message consisting of $B$ bytes. The cost of the above transpose procedure for a real $P \times Q$ matrix spread across $N$ processors is given by

$$T(P, Q, N) = \log N \ [4\frac{PQ}{N} t_{copy} + 2T_{comm}(2\frac{PQ}{N})] \tag{3}$$

The parameter $t_{copy}$ denotes the time to move one byte of data in the local memory. This data movement is implemented in FORTRAN by a pair of nested loops and can have an appreciable cost. Note that $t_{copy}$ will be smaller for complex arrays than for real arrays due to the lower loop overhead ***per byte.*** Unrolling the inner loop reduces the value of this parameter somewhat. An optimized variant of the transpose operation on a hypercube has been devised [3] that reduces local copying at the cost of more sends/receives and reduces the overall run-time.

A different approach is to use substructuring. By substructuring (SS) we mean reducing each system down to one equation per processor using what is essentially block Gaussian elimination in the manner described in [9], [4]. For substructuring, the number of floating point operations required per row is 17. In methods applying substructuring the load is perfectly balanced during this phase, and there is only one nearest neighbor communication between adjacent partitions in a subcube. We consider three alternative approaches for solving the reduced tridiagonal systems.

1. **SS/TGET**: This methods applies TGET to the reduced systems, after performing substructuring. The Gray code is not required for the transpose [3].

2. **SS/"Naive" CR**: Solve the reduced systems by cyclic reduction (CR), with all systems converging to the same processor. We do not recommend this method because of its poor load balancing properties.

3. **SS/BalCR**: The CR process is balanced, in the sense that an equal number of systems converge to each processor [5]. Each step of CR requires 17 floating point operations per row. The reduction phase and the substitution phase take $\log N$ steps each in CR. The number of rows modified in each processor is halved on successive steps. Each step in balanced CR requires shipping half as much data as the previous step, in contrast to the transpose, where the the amount of data shipped is the same for all steps. However, each step of BalCR requires twice as many start-ups as a step of the transpose. One potential advantage of SS/BalCR is the fact that for "sufficiently" diagonally dominant systems, the cyclic reduction process can be terminated in fewer than $\log N$ steps, where $N$ is the number of processors.

We now discuss the modifications that are required to implement the methods described above when $P < N$. Note that this regime is not important on small hypercubes such as the iPSC with 32 or 64 nodes, but will be relevant for larger cubes. To use more than $P$ processors, we must use two-dimensional domain decomposition. However, this involves separating the tridiagonal systems into groups that are confined to disjoint subcubes, and the problem on each subcube corresponds to one-dimensional domain decomposition, with more processors than systems. When $N > P$, TGET will involve moving one single system to each of $P$ processors, leaving the remaining $N-P$ processors idle while Gaussian elimination is being applied. This requires that the local memories be sufficient to hold an entire system. Even when sufficient memory is available, this approach will be unattractive when too many processors are idle. Similar considerations apply to SS/TGET, except that the local memory requirements are lower, since only the reduced systems are being solved by TGET. With SS/BalCR, after $\log P$ steps, we end up with single systems on subcubes of dimension $\log N - \log P$. We can then apply any method that is appropriate for solving single tridiagonal systems on hypercubes, for example cyclic reduction [5].

## 4.2 Comparison of Methods for Multiple Tridiagonal Systems

In the following, we compare TGET, SS/TGET and SS/BalCR. We first present the results of experiments on the iPSC and then use model times to predict the relative performance of the methods on cubes with different parameters.

We use the following simplified model of communication on the hypercube. The time to send or receive a message consisting of $B$ bytes is

$$T_{comm}(B) = Bt_c + \left\lceil \frac{B}{B_m} \right\rceil \tau,$$

where $\tau$ ($\approx$ 1.5 *msec*) is the start-up time, $t_c$ ($\approx$ 0.001 *msec*) is the transfer time per byte, and $B_m$ is the maximum packet size. $B_m$ = 1k bytes on the iPSC. This is different from the maximum message size, which is 16k bytes on the iPSC. This model neglects the difference between "internal" start-ups that occur between packets in the same message, and the "external", or initial start-up for the first packet. The considerable variability in the measured communications times for different runs with the same code and input parameters represents a significant difficulty in applying this model, or any other performance related model.

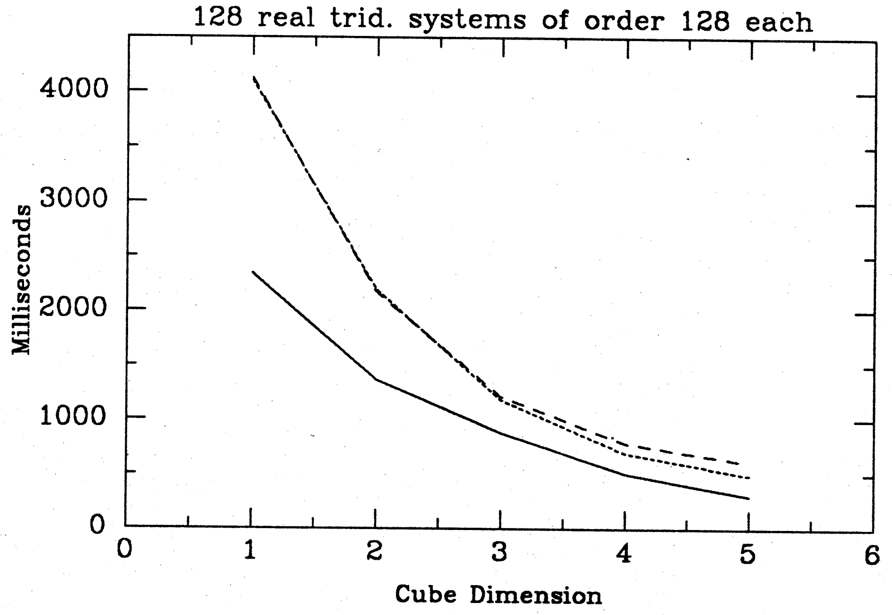

Figure 1: Solving 128 real tridiagonal systems of order 128 on the Intel iPSC with one-dimensional domain decomposition. Solid line: TGET, Dotted line: SS/TGET, Dashed line: SS/BalCR.

Figure 1 shows the time needed to solve 128 real independent tridiagonal systems, each of order 128 on the iPSC using one-dimensional domain decomposition, on cubes of dimension up to 5, using SS/BalCR, SS/TGET and TGET. Note that the amount of substructuring and the order of the reduced systems vary with the cube dimension, but all times are for solving the same problem. For this problem, TGET was the fastest, followed by SS/TGET, with SS/BalCR being the slowest.

We now discuss the relative merits and shortcomings of each method, to explain the ranking that is displayed in Figure 1, and why one can expect the ranking to change as parameters such as cube size, problem size, ratio of arithmetic to communication speed, etc. are varied. The advantage of TGET over SS/TGET is that SS/TGET has to pay the "substructuring penalty" of 17 floating arithmetic operations per row as opposed to 8 for Gaussian elimination in TGET. This is the primary factor that leads to the superiority of TGET for this problem. This advantage will be offset by the the fact that TGET transposes the entire data, whereas SS/TGET transposes only the data associated with the reduced systems as the problem size is increased. SS/BalCR also pays the substructuring penalty and in addition, requires twice as many start-ups as as SS/TGET (these two methods do not incur internal start-ups for this problem), which is why it is the slowest in Figure 1. However, since the amount of data communicated at each step is halved on successive steps for SS/BalCR and remains constant for TGET and SS/TGET, balanced cyclic reduction will be at an advantage for large cubes, and large problems. If the computational speed is increased relative to communication speed (e.g. by using vector boards), SS/BalCR and SS/TGET will benefit more than TGET because the substructuring penalty will be less crucial. Finally, reducing the copying time, $t_{copy}$, will help TGET and SS/TGET.

We state the expressions for the time required for each of the methods, in terms of the cube parameters.

$$T_{TGET}(P,Q,N) = 8\frac{PQ}{N}t_a + \log N\left(8\frac{PQ}{N}t_{copy} + 8\frac{PQ}{N}t_c + 4\left\lceil\frac{2PQ}{NB_m}\right\rceil\tau\right), \tag{4}$$

$$T_{SS/TGET}(P,Q,N) = 17P(\frac{Q}{N} - 1)t_a + 8Pt_a + \log N\left[20Pt_{copy} + 20Pt_c + 4\tau\right], \tag{5}$$

$$T_{SS/BalCR}(P,Q,N) = 17P(\frac{Q}{N} - 1)t_a + 17P(1 - \frac{1}{N})t_a + 80P(1 - \frac{1}{N})t_c + 8\log N\tau. \tag{6}$$

The expressions for SS/TGET and SS/BalCR have been simplified by assuming that messages do not exceed $B_m$ bytes, i.e. no internal start-ups are incurred and by neglecting the nearest neighbor communication performed in the substructuring phase.

Figure 2 compares the three methods using model times under the following assumptions: $P = Q$, $N \leq P$, $\tau = 1000$ and $t_c = 1$. This comparison assumes that sufficient memory is available. Each $(P,N)$ pair is represented by a box, which is shaded to indicate which method is fastest for that pair. The four plots, (A) through (D), represent different choices of the parameters $t_a$ and $t_{copy}$. SS/BalCR is the fastest for sufficiently large $P$ and $N$, but reducing the copying time shifts the crossover boundary in favour of TGET (or SS/TGET). Reducing the arithmetic cost (going from (B) to (C)), clearly favours the methods that apply substructuring. Note that plot (B) in Figure 2 corresponds approximately to the current parameters for the Intel hypercube.

## 5 ADM on Hypercubes

In this section, we discuss our implementations of the Alternating Direction Methods. We embed a two-dimensional processor mesh (torus) in the hypercube [6]. The $P \times P$ computational grid is mapped onto an $N_x \times N_y$ processor grid, with all processors receiving equal, contiguous blocks of grid points. $N = 2^n = N_x \times N_y$ is the total number of processors in the cube and $N_x$ and $N_y$ are powers of 2. We point out that for $N_y = 1$, this scheme reduces to embedding a linear array in the hypercube, or one-dimensional domain decomposition. For simplicity, we assume that $P$ is also a power of 2.

We recall that for Schrödinger's equation we use complex arithmetic. The tridiagonal matrix-vector products at each half step involve $C_{MVP}(P^2/N)$ arithmetic operations and nearest neighbor communication. The methods discussed in the preceeding section are central to the parallel implementation of the ADM. In going from real to complex arithmetic, with the grid size fixed, the arithmetic costs increase approximately by a factor of 5 and the data volume in each communication is doubled compared to the real case. Based on our experience with multiple real tridiagonal systems, where SS/BalCR was close to but slower than SS/TGET, we only report results for the ADM using the TGET and SS/TGET algorithms.

When we consider the complexity of a full ADM step, using a two-dimensional processor mesh, it is interesting to note that several terms in the final expression depend on $N$, but not on $N_x$ or $N_y$. These include the total arithmetic for the matrix vector products (MVP), for substructuring in SS/TGET and for Gaussian elimination in

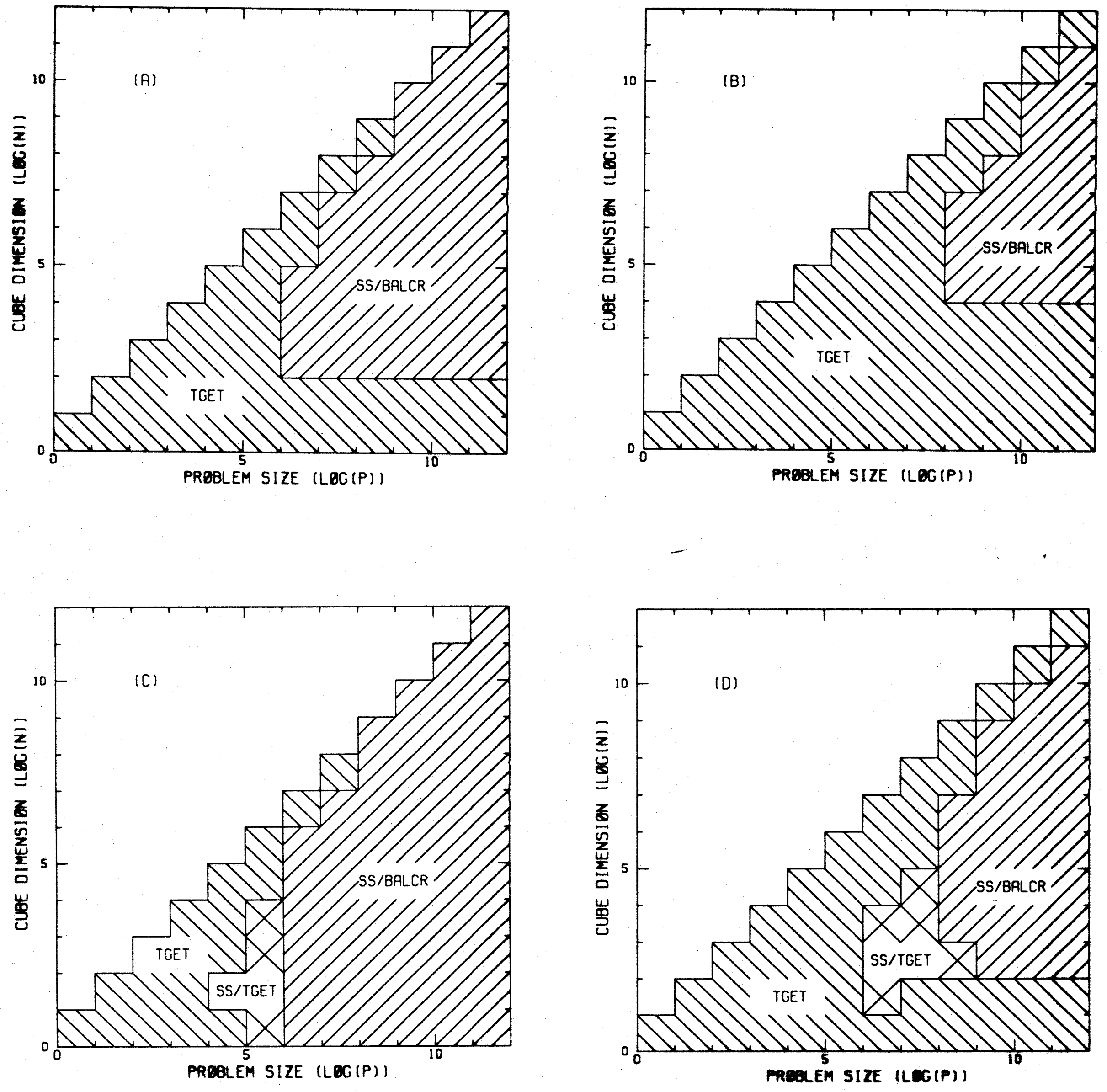

Figure 2: Comparison of TGET, SS/TGET and SS/BalCR based on model times for solving $P$ tridiagonal systems of order $P$, on $N$ processors. $N \leq P$, $\tau = 1000$ and $t_c = 1$. (A): $t_a = 25$, $t_{copy} = 10$, (B): $t_a = 25$, $t_{copy} = 5$, (C): $t_a = 5$, $t_{copy} = 5$, (D): $t_a = 5$, $t_{copy} = 1$.

TGET. Also, the nearest neighbor communications in the MVP and SS phases does not depend on $N_x$ or $N_y$, unless we are using one-dimensional domain decomposition. This phenomenon limits the sensitivity of the total time to the aspect ratio of the processor mesh. However, there are terms that depend on $N_x$ and $N_y$, and for $P \times P$ grids, these terms are minimized for $N_x = N_y = \sqrt{N}$.

We now derive the time for one ADM step using SS/TGET to give insight into the complexity of an ADM step. Let $C_{MVP}$, $C_{SS}$ and $C_{TRID}$ be the number of floating-point operations per point for the matrix-vector products, substructuring and the (local) tridiagonal solves respectively. The times for the different phases of a full step are

$$T_{MVP} = 2C_{MVP}\frac{P^2}{N}t_a + 2T_{comm}(8\frac{P}{N_x}) + 2T_{comm}(8\frac{P}{N_y}),$$

$$T_{SS} = 2C_{SS}\frac{P^2}{N}t_a + 2T_{comm}(32\frac{P}{N_x}) + 2T_{comm}(8\frac{P}{N_x}) + 2T_{comm}(32\frac{P}{N_y}) + 2T_{comm}(8\frac{P}{N_y}),$$

$$T_{RS} = (40\log N_x\frac{P}{N_y} + 40\log N_y\frac{P}{N_x})t_{copy} + C_{TRID}(\frac{P}{N_x} + \frac{P}{N_y})t_a$$
$$+ 2\log N_y[T_{comm}(\frac{16P}{N_x}) + T_{comm}(\frac{4P}{N_x})] + 2\log N_x[T_{comm}(\frac{16P}{N_y}) + T_{comm}(\frac{4P}{N_y})].$$

For tridiagonal matrices with complex coefficients with constants on the off-diagonals, $C_{MVP} = 16$ real arithmetic operations. Assuming that the matrices for the tridiagonal system solution have off-diagonal elements equal to $-1$, which is the case after scaling, $C_{SS} = 70$. For the reduced system we assume a general, complex, system, which for Gaussian elimination yields $C_{TRID} = 46$. This constant reduces to 24, if we assume that the sub- and super-diagonals entries are all equal to $-1$. With the simplified communication model, the total time is

$$T_{ADM}(P, N, N_x) = n_s\tau + (96 + 40\log N_y)\frac{P}{N_x}t_c + (96 + 40\log N_x)\frac{P}{N_y}t_c$$
$$+ (40\log N_y\frac{P}{N_x} + 40\log N_x\frac{P}{N_y})t_{copy}, + (172\frac{P^2}{N} + 46\frac{P}{N_x} + 46\frac{P}{N_y})t_a$$

where

$$n_s = 4\left\lceil\frac{8P}{N_xB_m}\right\rceil + 2\left\lceil\frac{32P}{N_xB_m}\right\rceil + 2\log N_y\left\lceil\frac{16P}{N_xB_m}\right\rceil + 2\log N_y\left\lceil\frac{4P}{N_xB_m}\right\rceil$$
$$+ 4\left\lceil\frac{8P}{N_yB_m}\right\rceil + 2\left\lceil\frac{32P}{N_yB_m}\right\rceil + 2\log N_x\left\lceil\frac{16P}{N_yB_m}\right\rceil + 2\log N_x\left\lceil\frac{4P}{N_yB_m}\right\rceil$$

is the total number of start-ups incurred, under our communication model. If we choose $N_x = N_y = \sqrt{N}$, $n_s$ decreases like $(\log N_x)/N_x$ as $N$ increases, but is bounded below by $12 + 4\log N$.

We now consider the case $N_y = 2$, which is a special case in the sense that when we apply TGET in the $y$ direction, each tridiagonal system is distributed over two processors. By using "two-way" Gaussian elimination, we can avoid the two transposes (which would have required just one step each) and replace them by one small exchange, involving just one row per system. Even though for larger cubes and larger problems with square grids, we expect a square processor mesh to be optimal, in our experiments we found $N_y = 2$ to be faster than $N_x = \sqrt{N}$. For a one dimensional cube, the tridiagonal solves in one direction are entirely local and this special case applies to the other direction For a two dimensional cube, this special case applies for both directions.

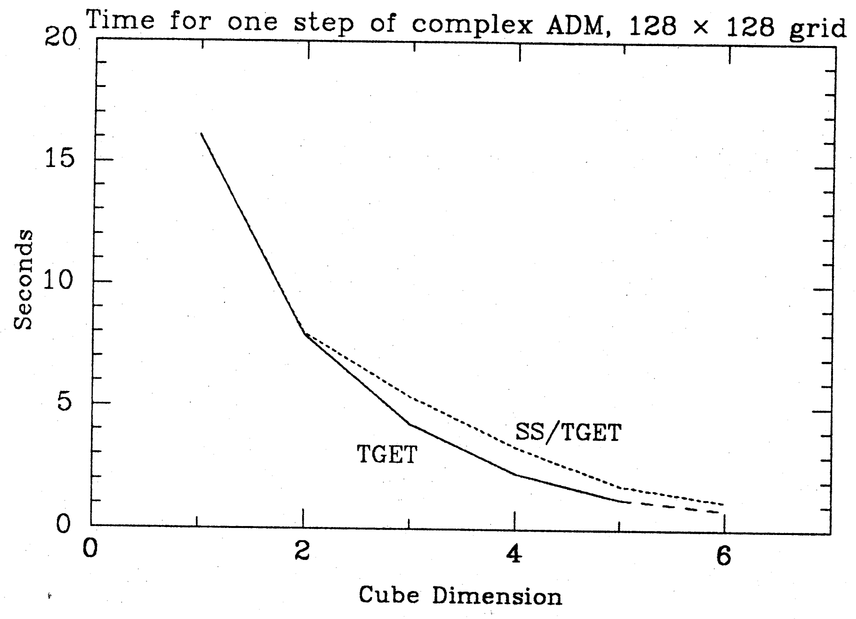

Figure 3: Cost of one step of ADM on the iPSC. Solid line: Using TGET (Dashed line: Extrapolation to 6-cube), Dotted line: Using SS/TGET.

## 6 Experimental Results on the Intel iPSC

In our experiments we used hypercubes of up to 5 dimensions. The floating-point capacity of each node is approximately 35 kflops (measured). The standard version of the Intel iPSC comes with 512k bytes of memory per node, but the Yale iPSC is currently configured with 4.5 Megabytes of memory per node. The Intel times reported here were obtained with the NX operating system, which has a reduced communication latency ($\approx$ 1.5 milliseconds) and the Ryan-MacFarland Fortran compiler, which handles complex arithmetic much more efficiently than the previous compiler.

In our experiments, we found that TGET with $N_y = 2$ was faster than SS/TGET for the 128 × 128 grid on cubes up to dimension 5 and the results reported correspond to this case, rather than $N_x = N_y$.

Figure 3 gives the time to perform one ADM iteration on a 128×128 grid, as a function of the cube dimension. With a 5-cube, TGET took 1.24 seconds, which represents a speed-up of 25.8 over the same code on a single node of the hypercube, and is faster than the VAX 8600 which took 1.4 seconds for the same problem. The code for TGET was not run on a 6-cube, and we have used an extrapolated value (0.8 seconds). The predicted time on a 7-cube is $\approx$ 0.6 seconds. Figure 4 shows the efficiencies corresponding to the times in Figure 3. The efficiencies are based on a running time of 32 seconds on a single node. For both methods, the efficiency falls off as the cube size is increased, but the deterioration is more rapid for SS/TGET because of the substructuring. Finally, Figure 5 shows the variation in the total time as $N_x$ is varied. As one can see, the "edge effect" of choosing $N_y = 2$ yields the best results for a 5-cube. TGET is less sensitive to variations in the aspect ratio of the processor mesh than SS/TGET.

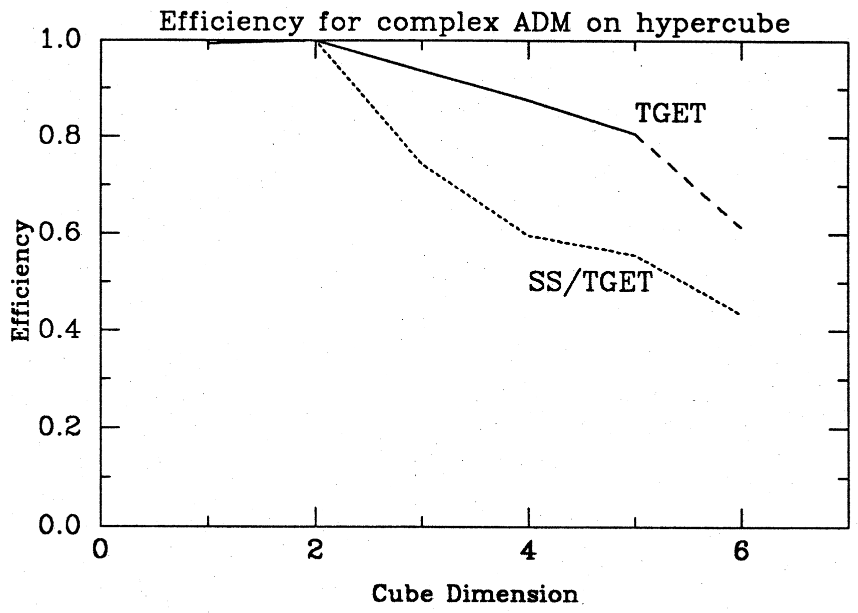

Figure 4: Efficiency of an ADM step. Solid line : Using TGET, Dotted line: Using SS/TGET.

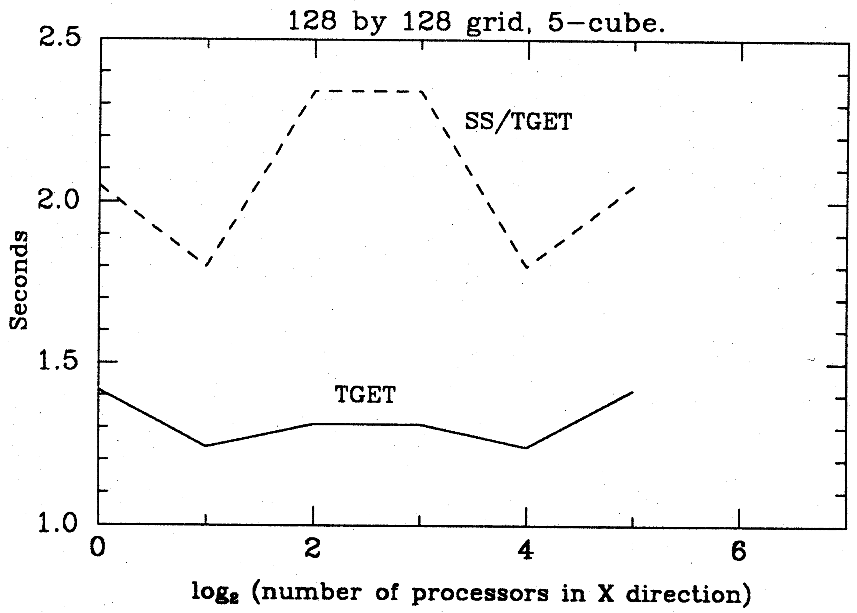

Figure 5: Effect of varying the aspect ratio of the processor mesh for a 5-cube. Time for one ADM step vs. $N_x$. Solid line: Using TGET, Dashed line: Using SS/TGET

## 7 Conclusion

The ADM scheme can be implemented efficiently on hypercubes. This is due to the fact that the method has regularity and a considerable degree of parallelism. The best iPSC time for a $128 \times 128$ grid is 1.24 seconds on a 5-cube, which represents a speed-up of 25.8, or an efficiency of 81%. Our predicted time on a 6-cube for the same problem is $\approx$ 0.8 seconds, which represents a speed-up of 40, or an efficiency of 63%. The time on a 7-cube is expected to be less than 0.6 seconds. The same problem takes 1.4 seconds on a VAX 8600.

The ADM scheme, due to its desirable numerical properties and the effectiveness with which it can be parallelized, can be recommended for the solution of Schrödinger's equation on hypercubes. However, future work on some of the other solution methods mentioned may lead to comparable or better parallel methods for the model problem.

**Acknowledgement**

This work was supported in part by the following: ONR grant N00014-84-K-0043, ONR grant N00014-86-K-0564, ONR grant N00014-82K-0184, NSF/CER grant DCR 8106181 and ARO FSP DAALO3-86-G-0029.

## References

[1] T.F. Chan, D. Lee, and L. Shen. *Stable Explicit Schemes for Equations of Schrödinger Type.* Technical Report YALEU/DCS/RR-305, Department of Computer Science, Yale University, 1984.

[2] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. *LINPACK User's Guide.* SIAM, Philadelphia, 1979.

[3] C.-T. Ho and S.L. Johnsson. *Matrix Transposition on Boolean n-cube Configured Ensemble Architectures.* Technical Report YALEU/CSD/RR-494, Yale University, Dept. of Computer Science, September 1986.

[4] S.L. Johnsson. *Odd-Even Cyclic Reduction on Ensemble Architectures and the Solution Tridiagonal Systems of Equations.* Technical Report YALE/CSD/RR-339, Department of Computer Science, Yale University, October 1984.

[5] S.L. Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. Sci. Stat. Comp.*, ():, 1986. Report YALEU/CSD/RR-436, November 1985.

[6] S.L. Johnsson, Y. Saad, and M.H. Schultz. Alternating direction methods on multiprocessors. *SIAM J. on Sci. Stat. Comp.*, ():, 1986. Yale University, Dept. of Computer Science, August, 1985, YALEU/CSD/RR-382.

[7] D.W. Peaceman and H.H. Rachford Jr. The numerical solution of parabolic and elliptic differential equations. *J. Soc. Indust. Appl. Math.*, 3:28–41, 1955.

[8] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms.* Prentice Hall, 1977.

[9] H.H. Wang. A parallel method for tridiagonal equations. *ACM Trans. Math. Softw.*, 7:170–182, 1981.