Figure 4 — An example of a rendered image (see article on page 54)
## Table des matières

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The ACL Message Passing Library</td>
<td>4</td>
</tr>
<tr>
<td>La Librairie d’envoi de messages ACL</td>
<td></td>
</tr>
<tr>
<td>James Painter, Patrick McCormick, Michael Krogh, Charles Hansen, &amp; Guillaume Colin de Verdière</td>
<td>4</td>
</tr>
<tr>
<td>O(N) Tight-Binding Molecular Dynamics on the T3D</td>
<td>10</td>
</tr>
<tr>
<td>Dynamique moléculaire O(N) Tight-Binding sur le T3D</td>
<td></td>
</tr>
<tr>
<td>Andrew Canning, Giulia Galli, Francesco Mauri, Alessandro De Vita &amp; Roberto Car</td>
<td>10</td>
</tr>
<tr>
<td>I/O Strategies on the T3D for CPU-Bound time dependent problems</td>
<td>17</td>
</tr>
<tr>
<td>Les stratégies d’entrées/sorties pour des problèmes dominés par le temps de calcul sur le T3D</td>
<td></td>
</tr>
<tr>
<td>Jean Latour</td>
<td>17</td>
</tr>
<tr>
<td>Monte-Carlo Simulation of Polymer Interfaces on the T3D</td>
<td>21</td>
</tr>
<tr>
<td>Simulation des interfaces entre polymères par méthode de Monte Carlo sur le T3D</td>
<td>21</td>
</tr>
<tr>
<td>Marcus Müller</td>
<td>21</td>
</tr>
<tr>
<td>Parallel Genetic Programming Induction of Binary Decision Diagrams</td>
<td>24</td>
</tr>
<tr>
<td>Induction par programmation génétique parallèle des diagrammes de décision binaires</td>
<td>24</td>
</tr>
<tr>
<td>Christian Miccio, Eduardo Sanchez &amp; Marco Tomassini</td>
<td>24</td>
</tr>
<tr>
<td>A Parallel Divide and Conquer EigenSolver</td>
<td>27</td>
</tr>
<tr>
<td>Calcul des valeurs propres et vecteurs propres par une méthode de Divide &amp; Conquer</td>
<td>27</td>
</tr>
<tr>
<td>Christopher P. Potter</td>
<td>27</td>
</tr>
<tr>
<td>Supercomputing: where have we been? where are we now? where are we going?</td>
<td></td>
</tr>
<tr>
<td>Superordinateurs: Où en étions-nous? Où en sommes-nous aujourd’hui? Où allons-nous?</td>
<td></td>
</tr>
<tr>
<td>Ron H. Perrot</td>
<td>32</td>
</tr>
<tr>
<td>4-D Quadrature Implemented in CRAFT</td>
<td>35</td>
</tr>
<tr>
<td>Implémentation CRAFT de la quadrature 4D</td>
<td></td>
</tr>
<tr>
<td>Robert M. Sinclair</td>
<td></td>
</tr>
<tr>
<td>Numerical Simulation of three-dimensional Time-dependent Buoyant Flows on the T3D</td>
<td>37</td>
</tr>
<tr>
<td>Simulation numérique d’écoulements tri-dimensionnels gravitationnels sur le T3D</td>
<td>37</td>
</tr>
<tr>
<td>Enrico Nobile, Luka Onesti, &amp; Marco Voli</td>
<td>37</td>
</tr>
<tr>
<td>On-line Visualization for Scientific Applications on the T3D</td>
<td>45</td>
</tr>
<tr>
<td>Visualisation en temps réel sur le T3D d’applications scientifiques</td>
<td>45</td>
</tr>
<tr>
<td>Steve Williams, Mark Sawley &amp; David Cobut</td>
<td>45</td>
</tr>
<tr>
<td>Operations of a T3D computer in an academic environment</td>
<td></td>
</tr>
<tr>
<td>Exploitation d’un système T3D dans un environnement académique</td>
<td></td>
</tr>
<tr>
<td>Jean-Michel Chenais</td>
<td>50</td>
</tr>
<tr>
<td>Parallel raytracing on the IBM SP2 and T3D</td>
<td>54</td>
</tr>
<tr>
<td>Lancer de rayons parallélisés sur l’IBM SP2 et le T3D</td>
<td></td>
</tr>
<tr>
<td>Igor-Sunday Pandzic, Nadia Magnenat-Thalmann &amp; Michel Roethlisberger</td>
<td>54</td>
</tr>
<tr>
<td>T3D experiments of an auto-tasked C90 code</td>
<td></td>
</tr>
<tr>
<td>Expérience de portage d’un code C90 avec autotasking sur le T3D</td>
<td></td>
</tr>
<tr>
<td>Bernadette Thomas &amp; Charles Rogé</td>
<td>58</td>
</tr>
<tr>
<td>Frenchess: A T3D at the 8th World Computer Chess Championship</td>
<td></td>
</tr>
<tr>
<td>Frenchess: un T3D au 8ème championnat mondial d’échecs</td>
<td></td>
</tr>
<tr>
<td>Marc-François Baudot, Jean-Christophe Weill, Jean-Luc Seret &amp; Michel Gondran</td>
<td>63</td>
</tr>
</tbody>
</table>
EDITORIAL

T his 8th edition of the EPFL Supercomputing Review, totally devoted to massively parallel technology, is a special vintage both in terms of quantity and quality.

After multiple efforts invested in this promising field, and following the preliminary and flourishing results of last year, the time of harvest has come. The 1995 vintage is a blend of multi-disciplinarity, performances and collaboration. The variety of the applications presented in this journal is as broad as ever: engineers and scientists provide statements about the major advances in computer science, material science, molecular dynamics, applied mathematics, parallel algorithms, visualization, image synthesis and fluid mechanics. The efforts invested are beginning to pay off, since the codes have reached the level of maturity necessary to make the most of the strongest characteristics of each MPP system considered (mainly T3D, CM5 and SP2). It should be noted that nine out of the fourteen articles of this edition were written by groups working outside of the EPFL, even though some of them used the EPFL computing facilities: this is an additional sign of collaboration, without which success would not be possible.

A special mention goes to the paper written by Ron Perrott, invited professor at the EPFL during this year. A number of interesting statements are made on the pitfalls frequently encountered using massively parallel systems; both the novice and the experienced programmer will be eager to read his concrete propositions.

Eleven articles, easily identified by a special logo, relate to presentations made at the First European T3D Workshop, jointly organized by the EPFL and CRAY RESEARCH in September, during which users of eight different T3D systems were present. An important pool of competences met at this workshop, and we are very pleased to be able to publish in these pages most of the contributions to this event.

Today, even though the illusions of easily accessible performance are left behind, the time has definitely come for the massively parallel technology to show its capability in achieving concrete results. A few years after the first pioneers started to explore the massively parallel technology, and eighteen months after the installation of the T3D at the EPFL, the experience and know-how gained by scientists and computer scientists are the most interesting return for investment. For the future, the directions that the EPFL may choose to pursue in the long term—such as developing the present scientific leadership, post- and under-graduate teaching, promotion of the applications and methods in industry—will rely mainly on these valuable human competences.

Producing such a journal would have been impossible without the faultless collaboration of the authors, the competence and patience shown by the two technical editors and the support of the Direction for Research at the EPFL: I personally thank each of them.

Marie-Christine Sawley
**THE ACL MESSAGE PASSING LIBRARY**

*by James Painter, Patrick McCormick, Michael Krogh and Charles Hansen, Los Alamos National Laboratory, USA & Guillaume Colin de Verdière, Centre d’Études de Limeil-Valenton, F-Villeneuve-Saint-Georges*

This paper presents the ACL (Advanced Computing Lab) Message Passing Library. Modeled after Thinking Machines Corporation’s CMMD, ACLMPL is a high throughput, low latency communications library for building message passing applications. The library has been implemented on the T3D, Thinking Machines CM-5, SGI workstations, and on top of PVM. On the T3D, benchmarks show ACLMPL to be 4 to 7 times faster than MPI or PVM.

**INTRODUCTION**

Parallel programs are typically written in one of two styles: SPMD or MIMD. SPMD (Single Program, Multiple Data) programs are typically written in a data parallel language, such as Fortran 90. With such programs interprocessor communication is hidden from the program via the compiler and runtime system. With MIMD (Multiple Instruction, Multiple Data) programs the programmer typically calls message passing primitives to perform interprocess communications. For such programs to run efficiently and gain the best possible speedup as additional processors are added, the communications cost of the program must be as small as possible. By small, we mean lowest latency and highest throughput. If communications costs are high, then the program will be severely limited in terms of speedup potential as the number of processors increases.

The development of the ACL Message Passing Library (ACLMPL) was driven by two motivations: performance and portability. As already mentioned, performance of a communications library is crucial to overall program performance. This fact was made all too clear when we started porting our message passing programs from the Thinking Machines Corporation (TMC) CM-5 massively parallel computer to the Cray Research Inc. (CRI) T3D. CRI supplies an implementation of the Parallel Virtual Machine (PVM) message passing library for the T3D; however, in our experience, its performance is poor. We found that our codes not only performed poorly but that they did not scale up (or run in some cases) to large numbers of processors. This was entirely due to the implementation of PVM on the T3D.

Our second motivation, portability, was driven by our investment in CM-5 software development. Many of the libraries and programs that we have developed for the CM-5 use a message passing library called CMMD. Upon the arrival of our T3D, we wanted an easy migration path for our software. Additionally, the CM-5 will still be a major production machine for some time to come. Thus having a common messaging system would be an added bonus.

The remaining sections describe previous message passing systems, describe the implementation of ACLMPL, present timings, describe a few applications that use ACLMPL, and draw conclusions.

**PREVIOUS WORK**

The PVM library was designed to treat a collection of computers, which may be workstations, servers, vector computers, or even MPPs, as a single distributed parallel computer [1]. To accomplish this, PVM supports heterogeneous processors, networks and data types. Besides basic communication primitives (asynchronous send and receive), PVM has primitives for process control, synchronization, signaling, process groups, and virtual machine control.

The Message Passing Interface (MPI) library was designed with efficiency and portability in mind. The MPI feature set was designed by a committee which used features and concepts from many various message passing systems [2]. What resulted is a “full-featured” message passing library that includes many variations on send and receive (blocking/nonblocking, buffered/unbuffered, receiver-ready, different data types including user specified, and more). Additionally, MPI includes support for global operations (barriers, reductions, gather/scatters, broadcasts, scans, etc.), processor topologies, processor groups, profiling, and error handling. Process management (creation, deletion, migration), active messages, and I/O support are not included in the current standard.

TMC created CMMD for the CM-5 massively parallel computer [3]. CMMD supports three styles of communication: synchronous, asynchronous, and active messages (used for event driven applications). The library also includes functions for global operations (reductions, scans, broadcasts, barriers) and parallel I/O. CMMD has no support for process control or virtual machine control.

Many other message passing systems provide similar functionality to these three. PVM, MPI, and CMMD are of particular interest to us since they are the “supported” message passing systems for the T3D and the CM-5.

**THE NEED FOR PERFORMANCE**

Our software efforts are targeted towards high performance software for MPPs and SMPs (Symmetric Multi-Processors). Our

SIMD (Single Instruction, Multiple Data) can be thought of as a more constrained SPMD.
focus is not on harnessing the latent power of desktop workstations. Nor is it in running a single program on several supercomputers. Given this, several key differences should be noted between PVM, MPI, and CMMD.

PVM is widely available for most Unix workstations and for many common supercomputers and MPPs. Portability, through support of heterogeneous data types and computers, is a main goal. PVM’s main weakness is that it is not high performance. One example of this is that past versions utilize a daemon process on each computer node which was involved in communications. Recent versions of PVM allow these daemons to be optionally by-passed; however, performance is still lacking as will be shown.

MPI is a recent message passing system and is not widely available at this time. MPI includes numerous primitives (far more than PVM), except for process management. While efficiency is a main goal for MPI, our benchmarks on the T3D show that it is lacking as well. MPI, like PVM, has the goal of supporting heterogeneous data types and computers.

CMMD differs from PVM and MPI in that it is not available on anything other than the CM-5; however, it does have a large user base since it was the only supported message passing system available on the CM-5 until recently (PVM was recently ported to the CM-5 by TMC). CMMD has a small set of primitives which are efficient, simple, but complete. It has the basic communications primitives as well as active messages. It also has the most commonly used global operations. CMMD was designed for interprocessor communications within the CM-5 and not with processes external to the MPP. This allows for several optimizations. Since the library is not designed to communicate with heterogeneous processors or data types, it avoids unnecessary data conversion and a plethora of different primitives for various data types. CMMD also takes advantage of the underlying hardware support. For example, it utilizes both the data network and the control network in the CM-5. In particular, the control network is used in global communications operations such as reductions and broadcasts.

ACLML was developed with similar constraints and goals as CMMD: message passing within a single multiprocessor machine (MPPs and SMPs) and sufficient primitives without trying to be all encompassing. As will be shown, this results in a message passing system, for both synchronous and asynchronous communications primitives, that is faster than PVM and MPI.

**IMPLEMENTATION**

ACLML is split into two groups: the synchronous communications primitives and the asynchronous primitives. On top of the synchronous primitives are layered the global communications primitives. Splitting synchronous and asynchronous primitives into two separate groups, with no overlap, allows for greater optimization than would be possible otherwise. For example, layering synchronous on top of asynchronous, will work but it introduces additional overhead (extra function calls, buffering, etc.). Additionally, the timings will show that synchronous communication can be faster than asynchronous communications.

The following sections will describe the implementation of ACLML on the T3D. Later sections will briefly discuss the CM-5 and SGI implementations.

**T3D SYNCHRONOUS COMMUNICATIONS**

The synchronous message passing API in ACLML was implemented first. Synchronous message passing has some potential performance advantages over asynchronous methods since there is no need for intermediate buffering. Data can be sent directly from the sender to the receiver with no need for additional data copying. This can result in much higher bandwidth and lower latency than is possible with an asynchronous protocol. The tradeoff is that computation cannot be overlapped with communications1.

A simple protocol built on the CRI SHMEM library shmem_put() function, which is faster than shmem_get(), is used as the lowest level communications primitive on the T3D [4]. Figure 1 shows the protocol used to send data between two processes on two separate Processing Elements (PEs). The receiving PE first writes a request block to the sending PE which contains the receive buffer address, its buffer length, and a control flag. The request block totals 16 bytes. Each PE has an array of request blocks, indexed by receiving PE. This avoids the need for locks on the request blocks since each block has only one writer.

The sending PE blocks, via a spin-wait loop checking the control flag, until this request block arrives. Once the request block is received by the sender, the sender initiates a shmem_put() from the local send buffer address to the receiver’s buffer address which is taken from the receive request block. Finally, after the data is transferred, a completion block is transmitted back to the receiver, indicating the size of the transfer, in bytes, and a flag value (DONE) indicating the transfer has completed. This completion block consists of 8 bytes.

The receiver, after initiating the request, waits in a spin-wait loop for its flag to change to DONE. Once the flag changes to DONE, both sender and receiver return. The synchronous protocol requires one round trip between the sending and receiving PEs and a total of 24 bytes of overhead information. This results in very low end-to-end latency (4.5 microseconds for a one word message transmitted between direct neighbor PEs) and high bandwidth (greater than 100MB/sec for one-to-all and all-to-one communication patterns).

Based upon the synchronous protocol there are three user callable functions: send, receive, and send_and_receive (send to one PE and receive from another PE, possibly the same).

**T3D ASYNCHRONOUS PROTOCOL**

Efficient asynchronous message passing exposes a number of implementation challenges on the T3D. Unlike the synchronous case, the asynchronous algorithm must address buffer management,
race conditions, and synchronization issues. Additionally, at least one extra data copy will be necessary between the application memory and a buffer within the message passing library, which is avoided in the synchronous case. Since word aligned `memcpy()` speeds on the T3D are only 170MB/sec (approximately), it is important to minimize the number of data copies in order to achieve high bandwidth.

Our approach to the buffer management problem follows that used in the Illinois Fast Messaging library [5]. As in FM, we use the fetch-and-increment registers on the T3D to allocate remote buffers from a fixed sized pool of buffers as shown in Figure 2. A sending PE reads the fetch-and-increment register on the receiving PE. The read operation returns the current value of the fetch-and-increment register, while atomically incrementing it as well. If the fetch-and-increment register is out of the bounds for the buffer pool, the sender must block until the receiver removes messages from the buffer pool and resets the fetch-and-increment register. If it is in bounds, the value read gives an index into the receiver’s buffer pool, providing a buffer which the sender has exclusive access to. The sender transfers the message data to this buffer, via `shmem_put()`, and transfers a flag value DONE, indicating the transfer is complete.

The receiving PE first checks a linked list of sent-but-not-yet-received messages for a message that matches the receive request. If a matching message is found, the data is `memcpy`d to the caller’s buffer and the linked list node is freed. If a matching message is not found in the linked list, the buffer pool itself is scanned for a matching message. If a matching message is found, the data is `memcpy`d to the caller’s buffer and the buffer pool slot is marked as RECEIVED. In most cases, the linked list is empty and a matching message is found directly from the buffer pool, resulting in a one data copy, in addition to the `shmem_put()`.

Each PE periodically checks whether its fetch-and-increment register has overflowed. This check is made each time a send or receive request is processed. The check can be accomplished by examining the last buffer in the buffer pool to see if it is marked as DONE or RECEIVED. If the fetch-and-increment register is out of bounds, all messages in the buffer pool are copied out into a linked list of sent-but-not-yet-received messages, and the fetch and increment register is reset to zero. This allows blocked senders to resume.

The user callable functions for the asynchronous protocol are asynchronous send, asynchronous receive, and blocking asynchronous receive. The two receives differ in that the first returns immediately if a message is not available. The other will block until a message has been received.

**Global Operations**

The global operations consist of a broadcast and a reduce primitive. The reduce primitive is extensible in that the user can write a reduction operator. Broadcast and reduce global operations are implemented in ACLMPL using efficient tree based algorithms [6]. For simplicity, both broadcast and reduce use PE 0 as the root processor, though the algorithms can be generalized to handle any root PE.

A broadcast from PE 0 is sent in $\log(P)$ phases, where $P$ is the partition size. In the first phase, only PE 0 is active and the broadcast is sent from PE 0 to PE $(P/2)$. In the second phase, PE 0 and PE $(P/2)$ are active and each sends to PE self + $(P/4)$. In the $i$th phase, PEs which have received the data forward the data onto the PE whose PE number differs only in the $(\log(P)-i)$th bit. This is a well known algorithm whose complexity is $O(N \log(P))$, where $N$ is the size of the broadcast and $P$ is the partition size$^2$.

The reduction operation uses the same tree structure used in the broadcast but in reverse, again yielding a $O(N \log(P))$ time bound. Initially all PEs are active. In the $i$th phase of the algorithm, the PEs which have a $1$ in the $i$th bit of their PE number send to the PE whose PE number is identical except for a $0$ in the $i$th bit. The sending node becomes inactive, while the receiving node combines the received data with its own and proceeds to the next phase. At the end of the reduction, PE 0 holds the entire reduced array.

Note that in each phase of the reduction, as we move up to the root of the tree, fewer PEs are participating in the operation. This suggests that a more efficient algorithm could be devised which utilizes all the PEs during every phase. We first made this observation in a special case of the reduction algorithm: image compositing in a sort last volume renderer [7]. In our binary-swap reduction algorithm we split the array being reduced in half at each phase of the algorithm and keep all PEs active throughout all phases.

In the $i$th phase of the algorithm, two PEs whose PE numbers differ only in the $i$th bit split their reduction array into two sub-arrays of equal size. One PE takes the lower sub-array while the other takes the upper sub-array. The two PEs exchange data, combine the received data with their own, and both proceed to the next phase. At the end of the reduction, the entire array has been reduced, but it is distributed across all the PEs. A final gather stage brings the result together in PE 0. The binary swap reduction algorithm runs in time $O(N \log(P))$ when the array size $N$ is much larger than the partition size $P$. On the T3D we have found that $N=1024$ is sufficient for binary swap reduction to outperform the simple tree based algorithm.

As previously mentioned, the global operations are built upon the synchronous primitives. Since all PEs must participate in a global operation, asynchrony is not needed. Furthermore, the synchronous primitives are faster since they do not do any buffering of data.

---

1. In earlier releases of the CRI `memcpy`, bandwidth performance was 10-30X worse!
2. Technically, this time bound and those that follow assume a hypercube interconnection network, though empirical evidence indicate that they match well to measured performance on the T3D 3D torus network as well.
ACLMLP for the CM-5

Since ACLMLP closely mimics CMMD, the CM-5 version of ACLMLP consists mainly of #defines instead of actual functions. This results in no overhead for using ACLMLP on the CM-5. The only real ACLMLP function is the reduction primitive. This is so that the user can write his or her own reduction operator, which is not supported by CMMD. Additionally, we have found the ACLMLP version of broadcast to be faster than the CMMD version for larger message sizes (approximately 2K bytes).

ACLMLP for the SGI

The Silicon Graphics version of ACLMLP is based upon IRIX specific interprocess communication (IPC) functions. These functions allow for the creation and management of a shared memory pool which is used to facilitate the communication of messages between processors. The current implementation lacks several optimizations, such as using direct memory mapping, which can increase performance. Future development will address this optimization and others.

In addition, ACLMLP has been implemented on top of PVM’s `psend()` and `precv()` functions. This not only provides us with a more portable version of the library, but can also help in the early stages of application development and debugging without the use of an MPP. We have found this to be particularly useful since the CRI debugger, Totalview, is not very stable.

Timings

Numerous benchmarks were performed on ACLMLP, MPI\(^2\), PVM, and SHMEM using the T3D. For each of these packages we attempted to write the most efficient programs possible. For example, in PVM we used `psend()` and `precv()` instead of using the other routines which pack and unpack the data. Six different test cases were run on various partition sizes and for various message sizes. The six cases are: one PE communicating with all others (one-to-all), all PEs communicating with all others (all-to-all), all PEs communicating with one PE (all-to-one), global reduction, global broadcast, and latency. Performance figures are included for SHMEM, in addition to the message passing systems, to give a reference for how they compare to using shared memory for communications.

The six cases were chosen for the following reasons. One-to-all is typical of initial data distribution, such as when one PE is responsible for reading a file and distributing parts of it to different PEs. Similarly, all-to-one is representative of gathering results back from all PEs for performing serial I/O. All-to-all is indicative of worse case, general communications. Global reduction and broadcast are included since they are very common global operations. The latency benchmark measures the overhead involved in sending very short messages (1 word) and measures the minimum overhead in sending short messages. Because many of the graphs exhibit similar curves, we have chosen a representative few for this paper.

Figure 3 and Figure 4 show the performance curves for the all-to-all case on 2 and 128 PEs. The Y axis shows throughput and the X axis shows message size in bytes. Several interesting features can be seen. Throughput for all of the message passing systems increases greatly until the message size becomes sufficiently large (greater than 1K bytes) and then tapers off. Synchronous ACLMLP is as fast as all of the other message passing systems for all cases. Additionally, for partitions greater than 2 PEs and for message sizes greater than 1K bytes, it is faster than either shared memory or the other message passing systems. This seems curious at first since ACLMLP is built on top of SHMEM. The explanation is that the SHMEM version floods the T3D network and causes collisions, thus reducing performance. Synchronous ACLMLP requires serialization (a PE can only receive from one sender at a time) which helps avoid saturating the network switches, thus resulting in greater performance.

The IRIX routines have better performance than the standard AT&T System V Release 4 IPC routines. See the SGI Insight manual “Topics in IRIX Programming”, for details.

\(^2\) The T3D MPI implementation was from EPCC. The MPICH implementation could not properly execute the test programs.
As the partition size increases, maximum throughput for the all-to-all case decreases from 67 MB/s to 23 MB/s. The kink in the PVM curve is due to a different, internal algorithm used by PVM for handling large messages. Finally, asynchronous ACLMPL functions are also faster than the other message passing systems for partitions containing 32 or more PEs. For partitions smaller than 32 PEs, ACLMPL is faster for message sizes less than 8K bytes.

Figure 5 shows performance curves for all PEs sending to one PE on a 128 PE partition. The synchronous version of ACLMPL is faster than the other message passing systems, as is the asynchronous version for messages less than 8K bytes. SHMEM is faster than ACLMPL in all cases since there is not the abundance of collisions on the network as there is with the all-to-all case. Maximum throughput is greater than 110 MB/s for synchronous ACLMPL.

The one-to-all case, Figure 6, exhibits similar performance curves with the exception that PVM seems to do better than it did in the all-to-one case.

Curiously, the spike in the PVM curve in the one-to-all case changes direction from all other test cases. Unfortunately, we have not been able to explain the direction change in the spike without access to the PVM source code for the T3D.

The one-to-all case, Figure 6, exhibits similar performance curves with the exception that PVM seems to do better than it did in the all-to-one case.

Figure 7 and Figure 8 show broadcast times for 2 and 128 PEs. Both graphs exhibit similar curves with the exception of the PVM curve. As the number of PEs increases, the upward spike in the PVM curve grows. It should also be noted, that as the number of PEs increases, the time for all message passing systems increases regardless of message size.

Figure 7 shows broadcast using 2 PEs.

Figure 8 shows times to perform a global reduce using 128 PEs. MPI is significantly slower than ACLMPL, and PVM performs well for small messages but then degrades for larger messages.

See the T3D PVM documentation on the PVM_DATA_MAX environment variable.
The ACL Message Passing Library

Table 1 shows the latency times for sending a one word message. Both the MPI synchronous and asynchronous versions incur significant overhead in sending a short message (greater than 8 times that of ACLMPL synchronous messages). It should be noted that the T3D is extremely instruction cache sensitive and that cache coherency and alignment will greatly affect these timings.

Table 1 — Performance for 1KB messages on 32 PEs

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Time (m seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACLMPL (sync)</td>
<td>5</td>
</tr>
<tr>
<td>ACLMPL (async)</td>
<td>15</td>
</tr>
<tr>
<td>PVM</td>
<td>25</td>
</tr>
<tr>
<td>MPI (sync)</td>
<td>47</td>
</tr>
<tr>
<td>MPI (async)</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 2 presents performance numbers for 1024 byte messages on a 32 PE partition which seems to be a commonly used size. The numbers for all-to-all, all-to-one, and one-to-all are in megabytes per second; and the numbers for broadcast and reduce are in seconds. For the first three cases, the synchronous functions in ACLMPL are approximately between 4 and 7 times faster than the other message passing systems, and broadcast and reduce are roughly 10% to 80% faster.

Table 2 — Latency

<table>
<thead>
<tr>
<th>Protocol</th>
<th>ACLMPL (sync)</th>
<th>PVM</th>
<th>MPI (sync)</th>
<th>PVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>alltoall</td>
<td>19.00</td>
<td>7.93</td>
<td>4.71</td>
<td>4.43</td>
</tr>
<tr>
<td>alltoone</td>
<td>61.80</td>
<td>36.43</td>
<td>10.70</td>
<td>9.70</td>
</tr>
<tr>
<td>one2one</td>
<td>74.00</td>
<td>60.61</td>
<td>10.70</td>
<td>9.48</td>
</tr>
<tr>
<td>broadcast</td>
<td>0.000076</td>
<td>0.000155</td>
<td>9.36</td>
<td>67.02</td>
</tr>
<tr>
<td>reduce</td>
<td>0.000182</td>
<td>0.000492</td>
<td>0.000138</td>
<td>0.000800</td>
</tr>
</tbody>
</table>

Table 2 — Latency

RESULTS

The application is used to simulate molecules containing several hundreds of millions of atoms. In 1993 it won a Gordon Bell prize for performance (it was able to sustain >53 Gflops on a 1024 node CM-5). It should be noted that at that time the application was based on CMMD. It is currently being ported to ACLMPL. ACLMPL has been used in two newly developed visualization applications. One is a sphere renderer that is used by the molecular dynamics project for displaying their data. The renderer can be used as either a standalone program or as a MIMD callable library. As a standalone program, the renderer can be used either interactively with an X11 graphical user interface (GUI) or in a batch mode. Images are either displayed in an X11 window, on a HIPPI frame buffer, or written to disk. Rendering rates on the T3D are approximately 660K spheres per second. For comparison, a SGI Onyx graphics workstation can sustain roughly 19K spheres per second.

The second visualization application is a renderer for volumetric data based upon Binary-Swap Compositing [8]. The renderer distributes a 3D data set to the PEs. Each PE is responsible for rendering its own subvolume. After each PE is done, the subimages are composited together using binary-swap. The user can interact with the renderer either through an X11 interface or through AVS. The renderer can generate approximately 4 frames per second using 128 PEs to render a 128³ data set into a 256 x 256 image that is displayed on a HIPPI frame buffer.

CONCLUSIONS

ACLMPL was developed with two goals in mind: to provide high throughput, low latency communications for message passing applications, and to provide portability. As previously shown, ACLMPL is approximately 4 to 7 times faster than either MPI or PVM on the T3D for general communications and 10% to 80% faster for global communications. This is significant to MPP applications since slow communications will reduce performance and scalability.

Since ACLMPL is based very closely on TMC’s CMMD, we can preserve our software investment. Additionally, we have found ACLMPL to be quite portable to other platforms while still retaining efficiency. While we don’t expect, nor want, ACLMPL to become the “new message passing standard”, we would hope that it can be seen as a challenge to those who implement message passing systems. ACLMPL should be viewed as proof that it is possible to develop a portable, usable, high performance message passing system for MPPs.

Finally, four major points should be noted. First, synchronous message passing is inherently simpler than asynchronous message passing. This is because buffer management and additional data movement can be avoided. These optimizations should be used. Second, efficient global communications algorithms exist and should be used; otherwise, scalability to large partition sizes is impaired. Third, on the T3D efficient buffer management can be performed using the fetch-and-increment facilities. Last, while portability is a highly desirable trait, perhaps performance should be equally important when supplying message passing systems for use within a MPP. MPI tends more towards this balance than does PVM, although additional performance gains should still be possible as we have demonstrated.
We present a parallel implementation of an O(N) tight-binding molecular dynamics algorithm on the T3D parallel computer. The localisation of the orbitals in the O(N) algorithm introduces a sparse nature to the orbital data and Hamiltonian which greatly changes the coding on parallel machines compared to a non-localised system.

The data distribution, communication routines and dynamic load balancing scheme used in the program are presented in detail along with the speed and scaling of the code on various homogeneous and inhomogeneous physical systems. Performance results will be presented for systems of 2048 to 32768 carbon atoms on 32 to 512 processors. We discuss the relevance to quantum molecular dynamics simulations with localised orbitals, of techniques used for programming short-range classical molecular dynamics simulations on parallel machines.

The linear nature of the scaling of the communications with system size and the localised nature of the orbitals makes these algorithms extremely scalable in terms of memory and speed on parallel systems with fast communications.
**Introduction**

![Image](https://via.placeholder.com/150)

**Figure 1:** Fullerenes ($C_{28}$) deposited on a reconstructed diamond (111) surface. The system shown has a supercell containing a 5072 atom slab of diamond with 52 $C_{28}$s deposited on the surface.

The evaluation of the Energy in Density Functional Theory (DFT) calculations using the Local Density Approximation (LDA), such as those used in Car-Parrinello type Molecular Dynamics (MD) simulations [1], requires of order $N^3$ steps where $N$ is the number of atoms in the system. This limits the size of systems that can be treated with these first-principles approaches to a few hundred, even with the most powerful modern computers. Tight-Binding (TB) models, while still presenting a quantum mechanical approach to molecular dynamics, have a much simpler basis set and Hamiltonian, greatly reducing the computational cost of an MD step. However, the scaling of the computational cost is still of $O(N^3)$ even though the prefactor is much smaller than for first-principles calculations. This limits the number of atoms that can be studied in TB MD simulations to about a thousand on the most powerful modern supercomputers. In order to extend the application of quantum MD to larger systems, many new, so-called $O(N)$ (since their computational cost grows linearly with the system size), techniques have been introduced in recent years. The basic ingredients of orbital based $O(N)$ approaches are the spatial localisation of the orbitals and a novel energy functional which does not involve explicit orthogonalisation of the orbitals or calculation of the inverse of the overlap matrix. It is well known that for certain systems e.g. insulators, the wavefunctions can be represented by exponentially localised orbitals (Wannier type functions), therefore, it should be possible to exploit this localisation in MD calculations.

In this paper we will discuss our approach to the parallel programming of the $O(N)$ energy functional proposed independently in references [2, 3] and [4] and extended to a non-orthogonal representation of the orbitals in [5]. However, a lot of the techniques presented in this paper are applicable to other $O(N)$ algorithms in the context of a TB model and to LDA implementations of $O(N)$ methods. We will not present in detail the calculation of all the different terms for our MD simulation but will rather present an overview illustrated with a few typical calculations such as the overlap matrix. The TB $O(N)$ scheme has already been used with a serial code to study fullerene impacts on a semiconducting surface [7] and one of the target problems for our parallel code is an inhomogeneous system of fullerenes deposited on a diamond surface (see Figure 1). In the section which follows we will discuss the implementation of the $O(N)$ TB MD in a serial code. In particular we will present how the Energy functional and its derivative are calculated in the context of sparse matrices. In the next section we will discuss the data distribution and communication routines used in the parallel code. In the following section we will look at techniques used in programming classical MD simulations on parallel machines that are relevant to our quantum simulations. In the section on dynamic load balancing, we will discuss its application to inhomogeneous systems such as fullerenes arriving on a diamond surface. In the last section we will give results for the speed and the scaling of the code for various systems.

**O(N) Tight-binding MD code**

Our code uses the conjugate gradient (CG) method to minimise the energy functional for the electronic degrees of freedom and the Verlet algorithm [8] to update the atomic positions in an MD simulation. The basic structure of the code is two large nested loops with the innermost loop performing the minimisation to the ground state of the wavefunctions and the outer loop the atomic dynamics. The minimisation of the energy functional in $O(N)$ calculations typically takes more steps than in non-localised methods and therefore a significant percentage of the calculation time for an MD run is spent in the CG minimisation step. Therefore, in this paper, we will devote most of the discussion of parallelisation and optimisation to the CG part of the code rather than the atomic update. In order to perform the CG minimisation it is necessary to calculate the energy functional and its derivative.

In the TB model the wavefunctions are expanded on the basis set of atomic orbitals at all the atomic sites. In the context of the $O(N)$ TB algorithm a localisation region for the orbital representation of the wavefunctions can then be identified with a set of atomic sites centered around a given atom. In our code atoms are included in a localised region (LR) if they belong to the $N_{is}$th nearest neighbour of the center atom where we have a cut-off distance to define, whether or not, an atom is a nearest neighbour. In this way, by varying the value of $N_{is}$ we vary the localisation region of the orbital.

The localised orbitals can then be represented by a set of coefficients

$$
|\Phi\rangle = \sum_j \sum_k c_{jk}^{i*} |\Phi_k^i\rangle,
$$

(1)

where $i$ is the atomic site on which the orbital is localised; $j$ is summed over all the atomic sites in the localisation region of site $i$; $k = 1,4$ the $p_x, p_y$ and $p_z$ atomic orbitals in our case) where $|\Phi_k^i\rangle$ is the $k$th basis function of site $i$. Outside the localisation region the value of the coefficients $c_{jk}^{i*}$ are zero. In a non-localised TB formulation, $c_{jk}^{i*}$ would be non-zero at all atomic sites. In the program the orbitals are stored as sparse coded matrices in the sense that we only store the non-zero values of
\( C_{ij} \): the values in the localisation region \( LR_i \), and a second matrix tells us which atomic sites each of these values corresponds to. In the program the complete set of orbitals for the system is stored as a four dimensional sparse coded matrix of the form \( C(k,m,l,i) \) where \( n \) is the index for the orbitals having the same localisation region and a two dimensional matrix \( \text{NEIG}(i,j) \) lists the true site numbers for each \( C \) value. A further vector \( \text{NNEIG}(i) \) contains the number of sites in each localisation region. It would be prohibitive, in terms of memory, and computationally inefficient to work with the full \( C \) matrix containing all the zero values of the \( C \)’s outside the localisation regions. It should also be noted that while for non-localised problems the overlap matrix, Hamiltonian etc. are usually calculated with Basic Linear Algebra Subprograms (BLAS), typically written in assembly language and highly machine-optimised, we now have to hand code the sparse matrix multiplications in Fortran. This leads to codes which typically have a MFLOP speed lower than non-localised codes but the \"time to solution\" for large systems is, of course much lower than standard \( O(N^3) \) non-localised codes.

In this paper we will discuss results for the code using the energy functional for non-orthogonal localised orbitals presented in reference [5]. This is essentially the same as the energy functional for localised orbitals presented in [2, 4] except that the number of orbitals is chosen to be larger than the number of electronic states. All the coding techniques we will discuss apply to both functionals. For more discussion on the physics behind these energy functionals the reader should consult these papers and the references therein. For more discussion on the physics behind these energy functionals the reader should consult these papers and the references therein.

The minimisation direction \( D^i \), which is conjugate to all the former directions of search, is then determined and the energy is minimised along the line \( C = C^i + t D^i \) where \( C^i \) is the current position. The energy functional is, in fact, a quartic in \( \chi \) (8th power if the Hubbard term is included) which can be calculated explicitly by evaluating the energy functional in equation 2 at \( C^i + t D^i \). Once \( t \) has been determined the \( n(C^i + t D^i) \) conjugate gradient process can then be calculated as \( C^i + t D^i \). This whole process is repeated until the required tolerance on the energy is reached. The majority of calculations to evaluate the quartic polynomial and the derivative of the energy functional involve sparse matrix products. We will take as a typical example the calculation of \( S \).

As previously noted \( S \) is stored as a four dimensional array where the first two indices correspond to the orbitals centered on the same site so that if there was no sparse coding \( S \) would be given by,

\[
S(m,n,j,i) = \sum_{l} C(k,m,l,i) C(k,n,l,i)
\]

where \( m = 1,3 \) and \( n = 1,3 \) are the indices for the 3 orbitals on each site; \( k = 1,4 \) is the index for the basis set of four orbitals and \( l \) and \( j \) are site indices. The sum over \( l \) is only non-zero on the sites where the two orbitals overlap. The type style \( C(k,m,l,i) \) is used to denote the way these variable are stored in the program. \( C(k,m,l,i) \) is sparse coded only on the \( i \) index while \( S(m,n,j,i) \) is sparse coded only on the \( j \) index. Thus in the calculation of \( S \) the indices \( m, n, k \) and \( i \) are direct indices in the sense that they run over all their possible values while \( j \) and \( l \) are indirect indices running over only the non-zero values of the matrices. The values of these indirect indices are given through our index matrices \( \text{NEIG}(i,j) \) and \( \text{NEIGS}(i,j) \). This makes our calculation different from standard sparse matrix multiplications which are typically of the form \( Z_{ij} = \sum_k X_{ik} Y_{kj} \) where only \( i \) is a direct index. In our calculation we are essentially pulling in two 3 x 4 matrices \( C(k,m,l,i) \) from indirectly indexed memory locations and multiplying them together to give a 3 x 3 matrix \( S(m,n,j,i) \).

\( m = 1,3; k = 1,4 \) from indirectly indexed memory locations and multiplying them together to give a 3 x 3 matrix \( S(m,n,j,i) \). Therefore in sparse notation \( S \) is given by, \( S(m,n,j,i) = \)

\[
\frac{\partial E}{\partial Q} = 4 \sum_{j} \left[ (h - h) \cdot (\partial Q)(2d_j - S_j) - \left| \partial Q \right| (h - h) \cdot \partial Q \right]
\]
\[
\sum_{\text{neigh}(a,b)} \sum_{(a,i)} C(k_n, a) \text{NEIG } (a, i) \text{NEIGS } (i, j) C(k_n, b) \text{NEIG } (b, j)
\]

where at each MD step we precalculate index lists giving the values of \(a\) and \(b\) for which \(\text{NEIG}(a,j) = \text{NEIG}(b,i)\) required for the outermost summation. Since we pull in small matrices rather than single values for each indirect memory reference these calculations run much faster on RISC type architectures with cache (such as the T3D) than a standard sparse matrix multiply. Using a data layout which gives the best memory access for the most common loops in our program we achieved speeds of about 30 MFLOPS for the calculation steps e.g. the calculation of \(S\).

The calculation of \(H | \Omega_i \rangle\), once we have constructed \(H | \Omega_i \rangle\), is of the same form as the calculation of \(S\) given in equation 6. The calculation of the quartic term \(\text{ZEIG}(a,j) \text{ZEIGS}(i,j) \text{ZEIG}(b,j)\) of the further calculation of \(S\) where \(D\) is the conjugate gradient direction. All these calculations are of the same form as equation 6, the calculation of the overlap matrix. In total, for a typical run, about 65% of the time in the CG step is taken up with sparse matrix multiplications while the remaining 35% is mainly vector matrix products (to calculate the derivative) which are also sparse coded. To perform the MD step the spatial derivative of the total energy \(E\) (including the ionic potential and kinetic energy) must be calculated. \(E\) is given by

\[
E = \frac{1}{2} \sum_i M_i R_i^2 - E_{\text{el}}(\{\Omega_i\}) - n_{\text{el}}(\{R_i\})
\]

and the ionic positions are updated using the Verlet algorithm [8].

\[
M_i R_i = -\frac{\partial E}{\partial R_i}
\]

and the ionic positions are updated using the Verlet algorithm [8].

Calculating the contribution to the forces from the electronic band structure term \(E_{\text{el}}\) takes the largest percentage of the time in the MD step and, as in the CG step, this calculation mainly involves sparse matrix multiplications. As part of the MD step we must also update the index lists \(\text{NEIG}(i,j)\), \(\text{NEIGH}(i,j)\) and \(\text{NEIGS}(i,j)\) as the sites within the different localisation regions may change. The direction cosines which occur in the matrix elements of the Hamiltonian must also be updated. In a typical MD run where 15 CG steps were performed for each MD step [4,7] the CG steps took 85 to 90% of the time.

**Molecular Dynamics on Parallel Computers**

It may be thought that the localisation of the orbitals in our Quantum MD calculation would lead to a system which is more similar in nature, from a programming point of view, to short-range classical MD than standard Quantum MD simulations. In this section we will discuss some of the techniques used in programming classical MD with short-range interactions which are relevant to our localised quantum MD calculations and we will also point out some of the major differences between these two types of simulations. In short-ranged classical MD simulations on parallel machines there are considered to be two main ways to distribute the data among the processors often referred to as particle and spatial distribution. In the case of a spatial distribution the cell in which we carry out the simulation is divided spatially into blocks which are then allocated to different processors. Each processor then stores all the data associated with the particles within its block (which may change during the simulation) and calculates the trajectories for these particles. In the case of a particle type distribution all the particles are divided among the processors (typically each processor dealing with the same number of particles) and throughout the simulation each processor will store the data associated with its set of particles (which do not change) and calculate the trajectories for its set of particles. With this type of distribution, if the particles are moving rapidly during the simulation (say in the case of a liquid or gas for example), there is no spatial correlation between the particles on the processors and their physical positions in the simulation. This means that when the data of a physically neighbouring particle is required, there is a high probability it will not be on the same processor and not even on a processor which is physically close. The particle data distribution can therefore lead to a high cost in communication between processors although it has the advantage that it is easy to load balance the calculations by giving roughly the same number of particles to each processor. On the other hand the spatial decomposition method can lead to large load imbalances in the calculations on each processor if the relative number of particles in each spatial region varies greatly. However, due to the spatial correlations between the particles on the processors and the true physical positions of the particles in the simulation it is highly probable that the data of neighbouring particles required to update a given particle will be on the same processor or at worst a physically close neighbouring processor. This means that a spatial distribution of the particles tends to be very efficient from a communications point of view but bad from a load balancing point of view. The best algorithms typically perform a combination of these two types of distribution giving roughly equal numbers of particles to each processor which have spatial locality and redistributing the particles to the processors, every few MD steps, if the particles move too much and lose their spatial locality. In this way both the constraints of load balancing and minimizing communication can be partially satisfied. It is an approach similar to this that we have followed for our Quantum MD program.

While we have drawn some analogies between classical short-range MD and quantum MD with localised orbital there are several major differences which will make our parallel implementation rather different from techniques used in classical MD. In particular, due to the minimisation of the electronic degrees of freedom, the amount of calculation required to update an atom at each MD step in quantum simulations is very much larger than the update of an atomic position for classical MD. This has implications for the loadbalancing and also the techniques used to calculate the localisation.

**Parallel tight-binding code**

In our program we used a particle/orbital distribution of the data among the processors so that each processor has complete orbitals. In the section on load balancing we will discuss how we...
implement spatial locality in the context of a particle like distribution for an inhomogeneous system. The reader may assume, in this section, that the code is written for a solid with periodic boundary conditions (such as bulk diamond) with the system being divided up into identical blocks, each processor dealing with the orbitals centered within this block and the atomic motion of the atoms in that block. We also assume the atoms are moving so little that they do not cross the boundaries of any of these blocks. For this system we have no load balancing problems and a spatial locality of the mapping of the orbitals onto the processors. We chose a particle/ orbital distribution (processors having complete orbitals) rather than a spatial type distribution (single orbitals being distributed between processors) for the following reasons. It is the simplest to program as the parallel code loop structures remain essentially the same as the serial code with a lot of the subroutines remaining almost identical. It gives faster calculation speeds on RISC type chips with cache than other methods. The communication cost of passing the orbital data between processors is modest due to the localised nature of the orbitals which means we only need to pass the orbital data for the overlapping portions of the orbitals which are not on the same processor. In our localised TB formulation the center of the localisation region is an atomic site so we can associate an orbital, as well as the atoms, to atomic sites. Therefore, each processor has a subset of atomic sites allocated to it and it will have stored in its memory all the data for the complete orbitals centered on that site plus all the atomic data for its sites. At each conjugate gradient step we require the values of the orbitals on other processors which are overlapping with the orbitals on our processor. We used a technique similar to ghost cells used in solving differential equations on parallel machines in that at each CG step each processor copies into a dummy array all the off-PE orbital elements that are required to update its own orbitals. Since during the CG steps the particles are not moving we can at the start of each MD step make a list of the processor numbers and memory locations of all the required off-PE orbitals. This list can then be used at each CG step to copy in the required parts of the orbital. It should be noted that during the conjugate gradient step, plus the conjugate gradient directions, must also be communicated between the processors. Our program is written in a message passing format using Fortran77 and PVM while the communication routines for the orbitals were written using the faster SHared MEMory (SHMEM) library routines which are native to the T3D. In addition to carrying all the local information for its set of particles and orbitals, each processor has global lists running over all the particles in the system which tell it which processor deals with a given particle and what is the local address of that particle. Using these global lists and its local neighbourhood lists each processor can determine the location of all its neighbouring orbitals. In addition each processor has a complete list of all the atomic sites so that the new neighbourhood lists can be calculated in parallel when the particles move.

**Dynamic Load Balancing**

There are now many articles on different load balancing schemes in short-range classical molecular dynamics applied to a range of different physical problems [10,13,14]. Load balancing becomes an issue for systems where the amount of time to calculate the new position varies from particle to particle. Giving out the same number of particles to each processor would then cause processors to idle while waiting for the processor with the most work to finish. As discussed in previous sections, the main problem in developing a load balancing algorithm to divide the calculations among the processors is to satisfy the (often conflicting) constraints of spatial locality and the calculations between processes. If particles/orbitals are rapidly moving during the MD simulation any load balancing algorithm which does not take into account spatial locality will lead to increased communications during the run. The most sophisticated load balancing algorithms for classical MD simulations typically involve division of the system into a number of different sized regions equal to the number of processors such that the calculations, for all the particles in each region, take roughly the same time [12]. These regions should then have a surface area to volume ratio as low as possible in order to minimise the amount of communications. Algorithms to perform this type of load balancing are often difficult to implement in three dimensions and can have numerical instabilities and bad convergence. In most algorithms the region that each processor deals with is topologically connected. In our load balancing algorithm we will weakly relax this constraint while strongly satisfying the load balancing constraint on the calculations. In our typical simulations the communications only take about 10% of the time so that even if the spatial locality was strongly satisfied we would only expect a few percent difference in the program run time between our solution and the optimal solution. On the other hand the calculations take about 90% of the run time therefore good load balancing is extremely important.

The first step in our load balancing algorithm is to spatially divide the system into two or three dimensional blocks. Typically the number of blocks is chosen to be equal to the number of processors but this is not a necessary condition for our load balancing algorithm. In our program we read in three parameters which correspond to the number of blocks in the x,y and z directions. We then construct a one dimensional list of all the sites by ordering the blocks as a one dimensional list with x,y,z ordering and each block we write as a one dimensional list with y,z,x ordering. It is important that the ordering of the sites for each block is the reverse of the block ordering to give good spatial locality although it is not necessary to order the blocks as x,y,z. There may be systems where better spatial locality is obtained by ordering with the y or z coordinate being the most rapidly varying. In this way we construct a one dimensional list that has three dimensional spatial locality. We now wish to load balance on this list giving out the sites to each processor such that each processor does the same amount of work. As discussed in previous sections, the main problem in developing a load balancing algorithm to divide the calculations among the processors is to satisfy the (often conflicting) constraints of spatial locality and the calculations between processes. If particles/orbitals are rapidly moving during the MD simulation any load balancing algorithm which does not take into account spatial locality will lead to increased communications during the run. The most sophisticated load balancing algorithms for classical MD simulations typically involve division of the system into a number of different sized regions equal to the number of processors such that the calculations, for all the particles in each region, take roughly the same time [12]. These regions should then have a surface area to volume ratio as low as possible in order to minimise the amount of communications. Algorithms to perform this type of load balancing are often difficult to implement in three dimensions and can have numerical instabilities and bad convergence. In most algorithms the region that each processor deals with is topologically connected. In our load balancing algorithm we will weakly relax this constraint while strongly satisfying the load balancing constraint on the calculations.

The first step in our load balancing algorithm is to spatially divide the system into three dimensional blocks. Typically the number of blocks is chosen to be equal to the number of processors but this is not a necessary condition for our load balancing algorithm. In our program we read in three parameters which correspond to the number of blocks in the x,y and z directions. We then construct a one dimensional list of all the sites by ordering the blocks as a one dimensional list with x,y,z ordering and each block we write as a one dimensional list with y,z,x ordering. It is important that the ordering of the sites for each block is the reverse of the block ordering to give good spatial locality although it is not necessary to order the blocks as x,y,z. There may be systems where better spatial locality is obtained by ordering with the y or z coordinate being the most rapidly varying. In this way we construct a one dimensional list that has three dimensional spatial locality. We now wish to load balance on this list giving out the sites to each processor such that each processor does the same amount of work. As essentially all the subroutines in the program the outermost loop is over the sites allocated to each processor. In a typical subroutine, such as the calculation of the overlap matrix, we explicitly time the calculations associated with each site by adding a subroutine timing call within the loop over the sites. We now have a time \( t_i \) associated with the calculations for each site where \( i = 1, 2, ..., N \), the order of sites being the one dimensional list we have constructed with spatial locality. We now calculate the time each processor would take to do the calculations if the system was perfectly load balanced i.e.

\[
T_y = \sum_{n=1}^{N} t_i
\]

where \( n_{\text{proc}} \) is the number of processors. We then hand out contiguous strips of sites to each processor, from our one dimensional...
list, such that the sum of the times \( t_i \) on each processor are as close as possible to \( \sum_{j=1}^{n} t_j \). In this way we achieve an essentially optimal load balancing while still having a high degree of spatial locality of the mapping of the sites onto the processors. This means \( m \) sites, \( j, j + 1, j + 2, \ldots, j + m \) are allocated to a given processor such that,

\[
\sum_{i=j}^{j+m} t_i \equiv T_{sp}.
\]  

Providing the the ratio of the number of atoms to the number of processors is not too small, we have found this load balancing algorithm to work extremely well. In a typical simulation where we had \( \approx 40 \) sites per processor we achieved load balancing down to a few percent. We perform the load balancing dynamically during our MD run every 10 to 20 MD steps depending on how rapidly the atoms are moving in our simulation. All the data for the orbitals and the atoms is reorganised in memory to correspond to the new mapping of sites to processors. Due to the computationally intensive nature of quantum MD the load balancing typically takes less than one percent of the run time if it is carried out every 20 MD steps. This should be contrasted with classical MD where the load balancing can easily become a significant percentage of the run time. It should be noted that our load balancing algorithm is very general in that the same program can run any geometry of system, even systems of different dimensions, on any configuration of processors.

**Performance**

In this section we will present performance results for an idealised bulk diamond systems and for the inhomogeneous system of fullerenes arriving on a reconstructed diamond surface which was discussed in the previous section on load balancing. The inhomogeneous system used for these tests consists of 52 fullerenes, each of 28 carbon atoms, deposited on a diamond slab (3072 atoms) of 12 double layers, with periodic conditions in the \( x \) and \( y \) directions (see figure 1). The performance figures were taken for a typical dynamical run, with dynamic load balancing every 20 MD steps, where 5 fullerenes are deposited on top of 47 fullerenes already bonded to the diamond slab.

The bulk diamond system used in the tests has periodic boundary conditions in the \( x, y \) and \( z \) directions. As mentioned in the section on the parallel tight-binding code each processor deals with a geometrically equivalent sub-block of the diamond system. During the MD runs on bulk diamond the atomic motion was constrained so that the neighbourhoods defining the localisation regions remained unchanged throughout the calculation. The calculation times for constructing the index lists at each MD step are still included in the performance figures to make them more representative of an MD run on a real system in a research environment. Due to the static and homogeneous nature of this problem no load balancing was required or performed during these runs.

![Figure 2: « Weak scaling » of the code for a bulk diamond system of 2048 to 32768 atoms running on 32 to 512 processors where the localisation region for each orbital contains 45 sites. The curve of circles shows the code performance for the conjugate gradient minimisation and the curve of squares shows the performance for a full molecular dynamics run.](image)

In figure 2 we show the results for the « weak scaling » of our code where the size of the system is increased proportionally with the number of processors. The results shown are for a 2048 atom bulk diamond system on 32 processors up to a 32768 atom system on 512 processors. The localisation region is taken up to the 3rd nearest neighbour giving 45 atomic sites in each localisation region. This is a localisation region which is sufficiently large to give good convergence and very small errors (compared to a non-localised algorithm) for carbon based insulators. The results for the MD runs are given with 15 CG steps for each MD step which is representative of the amount of CG steps required for convergence in a typical run [4,7] for this type of system. As can be seen from the graph the speed up is extremely linear and very close to ideal. The full MD simulation MFLOP speed is slightly lower than the CG step due to the index lists that are constructed at each MD step. Also some of the calculations for the atomic update, run at a slightly lower MFLOP speed than the CG step. We present the results in GFLOPs rather than seconds for an MD step as the smaller systems have wraparound effects on the localisation regions which means that proportionally less time is required for the MD step than larger systems. It should be noted that the algorithm used is strictly \( O(N) \), in terms of the number of calculations per MD step, so that provided there are no wraparound effects on the localisation regions the speed up in GFLOPs translates directly into an identical increase in the number of atoms updated per second. All of these simulations spent about 14% of the time in communications and 86% doing calculations. An MD step, including 15 CG minimisation steps of the orbitals, took about 50s for the 32768 atom system on 512 processors, running at above 11 GFLOPS. The weakscaling curves for other ratios of particles to processors and different localisation regions are also essentially linear with nearly ideal speedup. Due to the localised nature of the problem the ratio of communications to calculations remains the same as we scale up the problem and the communications remain local. The GFLOP speed for the inhomogeneous system on 128 processors is \( \approx 8\% \) lower than the
bulk diamond system. The lower GFLOP speed is caused by two main factors the first of which is the load imbalance introduced into the problem by the inhomogeneous nature of the system and the fact the particles are moving. The second most important factor is the increase in communications due to the less spatially structured nature of the problem which makes the spatial mapping of the system to the processors less direct than the bulk diamond system. A third less important factor is the extra time required to run the load balancing algorithm every 20 MD steps although this is only about 1% of the program time. Without dynamic load balancing our test run is about twice as slow, which is due to the fact that when the fullerenes arrive at the surface the number of sites in the localisation regions of their atoms can increase by factors of 2 to 3.

These results should be compared with the vector code that runs at about 270 MFLOPS on one processor of a C90 for the bulk diamond system. The number of floating point operations per MD step is, to within a few percent, the same for the vector and serial code. It should be noted that the structure of the loops in the vector code is very different from that of the parallel code. The innermost loop in the vector code must be over sites since this is the only loop long enough to efficiently exploit the vector processing while in the parallel code we parallelise over the site loop which must therefore be the outermost loop.

Figure 3: — « Strong scaling » of the code for two bulk diamond systems of 2048 and 4096 atoms, and the inhomogeneous system shown in figure 1. All orbital localisation regions are taken up to the third nearest neighbour.

Figure 3 shows the « strong scaling » performance for the code where the size of the system is kept constant and the number of processors is increased. Results are shown for 32 to 256 processors for the 2048 and 4096 atom bulk diamond system and also the inhomogeneous system. The choices of systems are meant to be typical of the size of problem that would be run in a research environment on a system of 32 to 128 processors. The speed up is very linear and in the case of 4096 atoms of diamond there is a 7.3 speed up in going from 32 to 256 processors. The speed up is lower for the smaller 2048 atom system since when we are in the regime where the number of atoms on each processor is very small (8 atoms per processor for 256 processors), the amount of time in communications becomes more important. This is because the fewer atomic sites we have on each processor the more probable it is that the orbitals, required to perform the calculation of the overlap matrix elements etc. for these sites, will not be on that processor and will have to be communicated. Typically the communicated data will also be less local on the processors when we have a few atoms on each processor thus, the communication times will be larger. In order to reduce the communication cost of studying small systems on a large number of processors it may be more efficient to use a different data distribution where the data for each orbital is spatially distributed over the processors. However, it is not clear that the overall speed would be faster as the MFLOP speed for the calculations may decrease. This type of data distribution was implemented in reference [15] on a CM5 but the single PE performance is very much lower that what we find for our implementation. The load balancing also becomes more difficult as we cannot simply load balance over sites since the calculations associated with the MD step for each individual atom are distributed over processors. All our production runs of the code were never in the regime where there was a few sites on each processor which would only be useful for studying small systems over very long timescales.

CONCLUSIONS

In this paper we have presented a particle type data distribution scheme of an $O(N)$ TB molecular dynamics scheme. We have shown that the local nature of orbitals translates directly into a parallel algorithm that has only local communications between processors. While the amount of interprocessor communications in our program is significant the locality of these communications leads to an algorithm whose ratio of communications to particle number remains constant as we scale up the processor number and system size. We have also presented a simple and robust load balancing scheme for load imbalance found in inhomogeneous systems or systems with rapidly moving atoms. Parallel machines with fast communications, like the T3D, are very cost-effective for these types of simulation as well as allowing the possibility to scale to larger systems than could be studied on conventional supercomputing platforms. Our program has now been used to study many different problems e.g. fullerene deposition, in a variety of carbon based systems. The results of these simulations and a more detailed description of the code will be published elsewhere.

ACKNOWLEDGEMENTS

This work was done on the T3D at EPFL as part of the PATP (Parallel Applications Technology Program) joint project between the EPFL and Cray Research. Support is also acknowledged from the Swiss National Science Foundation. In the course of this work we have benefited from useful discussions with other PATP collaborators and staff at Cray Research, Eagan, USA.
I/O STRATEGIES ON THE CRAY T3D FOR CPU-BOUND TIME DEPENDENT PROBLEMS

by Jean Latour, Department of Services, Cray Research, France

L’organisation des Entrées/Sorties, pour des problèmes dominés par le temps de calcul, tels que les résolutions d’équations aux dérivées partielles, peut se faire sur le T3D avec la notion de noeud spécialisé: seuls quelques processeurs traitent l’ensemble des entrées/sorties nécessaires à tous les autres, la redistribution des données étant faite par le réseau interne au T3D qui est beaucoup plus rapide que les canaux externes vers les disques. Les tests présentés ici montrent que, pour une quantité de données fixée, le nombre de noeuds d’E/S influe assez peu sur le temps global des entrées/sorties. Il est donc préférable de ne charger qu’un nombre assez restreint de processeurs d’effectuer les E/S (1 à 8 par exemple). De plus, la zone mémoire utilisée par l’agent (processus chargé des requêtes d’E/S sur la machine frontale) doit être suffisamment importante pour garder une bonne performance lorsque le nombre de requêtes d’E/S augmente. D’autres facteurs pouvant influer sur les performances telles que la longueur des enregistrements, ou l’utilisation des bibliothèques de communications internes au T3D sont brièvement discutées (SHMEM ou PVM). L’ensemble des tests a été fait en FORTRAN standard, avec access séquentiel ou direct sur un ou plusieurs fichiers.

For CPU Bound, PDE’s problems, we investigated the «pipe node» I/O strategy where a few PE’s are performing the I/O necessary to all PEs. Data redistribution is made through the T3D network whose bandwidth is much larger than those of the disk channels. The tests presented here show that, for a fixed amount of data, the I/O wait time is only slightly dependent on the number of pipe nodes. Usually, one to eight pipe nodes may be sufficient for a good performance. One key factor is the memory size of the agent process, who controls I/O on the前端 computer. A large memory size is needed when the rate of system calls increases. We discuss briefly how the length of records or the T3D communication libraries (shmem or PVM) can influence the overall performance. All tests are done with standard FORTRAN, sequential or direct access on one or several files.

INTRODUCTION

When solving 3D time dependent PDE’s on a massively parallel machine, the organization of the data flow in and out of the distributed memory can be a major issue. Applications that have to solve this problem are for example: weather forecasting, climate studies or oceanographic modeling. Reading the initial data and writing timely samples of the solutions without hampering the overall performance of the code requires some tuning. The aim of this paper is to give qualitative and quantitative data that may help to write such applications, using the concept of “pipe nodes” [figure 1].
if the « one file per PE » strategy is probably the easiest to code, applications that can use a large number of PEs should give the I/O capability to only a small subset of them: the “pipe nodes”. These will have the additional burden to collect or distribute the relevant data between the disk file(s) and a subset of PE’s memories. Non-pipe nodes will just exchange data with the pipe nodes through the T3D Network (see figure 1). This last operation can be done concurrently with the computations using the SHMEM library.

For a given amount of data and a given number of PE’s (NSPES), there are several free parameters to adjust: the number of pipe nodes, the number of files, and the length of records are the most obvious. In order to help this tuning, we report three I/O experiments:

1) Reading a single shared Fortran file through Direct Access by the PE's. The total amount of data is fixed, but the number of pipe nodes (and the number of records per pipe node) vary. These experiments have been carried out in a “normal” environment: the T3D was not a dedicated machine, and the system parameters were not modified. Quantitative measurements of I/O wait time in these conditions must be taken very cautiously: they are not the fastest time one can obtain on a given hardware and they can vary by a large factor when repeating the same experiment. The times reported here give an order of magnitude of the I/O wait time. More important are the variations of these times when varying the free parameters: the number of pipe nodes and the number of records per pipe node.

We carried also some experiments allowing a larger memory size for the agent process and found that this factor can be critical for the overall I/O performance.

The I/O process on the T3D

Many hardware I/O configuration can be specified on T3D computers. Detailed descriptions of them, known as phase I, II or III are given by [1]. We just recall here a few characteristics of the basic phase I configuration.

a) Control of the I/O process is done by the “agent” process on the front end computer: it executes the I/O system calls issued by the microkernel of the PE’s. The agent forks himself several times in order to control simultaneously a given number of PE’s. By default the “mppexec” process controls 4 PE’s, but this number can be modified through the environment variable MPP_AGENT_SYSCALL_THREADS. The overall memory size used by these agent processes can be critical for the I/O performance. It can be controlled by several environment variables, as described by [2]. Examples will be given below.

b) The data flows from the disk to the I/O buffers of the front end computer, and then to the T3D's Master I/O gateway(s) through one (or more) highspeed 100 Mbytes/sec channel(s).

c) The data then flows from the MIOG to a given PE through the T3D network. Actual measurements of the continuous data rate through the T3D network is accurately described in [3].

d) The slowest part of the I/O process is the disk channels on the front end: sustained rates are usually between 10 to 20 Mbytes/sec. In order to use these channels efficiently, long records and large buffers are recommended; the faster the device, the longer the records; typically a record length of several megabytes should be preferred.

Private I/O with Fortran

The T3D software allows for private or shared I/O. All the tests presented here are done with private I/O and FORTRAN. Shared I/O can be more easily performed in C language, and requires explicit synchronization between the PE’s. With shared I/O the file positionning done in one PE is relevant to other PE’s. With private I/O file positionning is local to each PE, but the PE’s can open a single common file or different files. For a write operation in FORTRAN one must use different private files for each pipe node since the user cannot control the synchronization during the disk transfers. For a read operation, several PE’s can concurrently access the same portion, or different parts of a single file. Every pipe node can open the same file on a private unit number. Direct Access or sequential Access allows to read this file simultaneously from several PE’s. We have done I/O tests using this method, and also using different files on different PE’s.

Simple I/O Experiments using Pipe Nodes

Four experiments are reported hereafter, measuring the I/O wait times in each PE. For a given amount of data, we vary the number of pipe nodes and the number of records per pipe node.

Reading a Single Fortran Direct Access File

During initialization of the solution of PDE equations, every computing node must receive specific data. One way to do this is for each pipe node to open (privately) the same file and access to a specific record. Using Direct Access in this context is more efficient than sequential Access since, in the later case, the PE that needs the last part of the file has to read all of the records. Instead, with Direct Access the file positionning of each pipe node is much shorter, and some data already in the I/O buffers can be re-used by several PE’s. After reading some specific records the pipe nodes then send to a subset of PE’s the relevant data, for example one record per PE. We give quantitative data on this experiment on tables (1) and (2). The numbers refer to the I/O wait time in seconds measured in each PE with the irtc() function. For each case we give the minimum and maximum I/O time. This time increases within a subset of PE’s depending of one pipe node with the record position in the file, reflecting just the time needed to read consecutive records.
For pipe nodes the real clock time measured is:

\[
t_1 = \text{irtc}(())
\]

open the file
do i = 1, N$PES/npipe
    read one record
    call pvmfpsend (to other nodes)
enddo
close the file
\[
t_2 = \text{irtc}()
\]

print *, t2 - t1

For non-pipe nodes we also include the wait time for the data to reach the PE, by measuring the time for: open + pvmprecv + close.

Of course in a real problem the open and close are not necessary and only on the pvmprecv will some time be wasted. In fact the T3D network can be used more efficiently with shmem - put done by the pipe nodes. In this way the non-pipe nodes are not interrupted during the data transfer. A synchronization point is however needed before the computations resume with the new data.

Table 1 refers to N$PES = 32, record length is always 100 000 words (.8 Mbytes), and FORTRAN Direct Access to a single file is done by npipe pipe nodes.

Table 1

<table>
<thead>
<tr>
<th>NPIPE</th>
<th>RECORDS PER PIPE NODE</th>
<th>T: STANDARD AGENT</th>
<th>T: AGENT MEMORY SIZE = 30 Mw</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>.2 =&gt; 6.9</td>
<td>.7 =&gt; 5.4</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>.0 =&gt; 5.4</td>
<td>.4 =&gt; 5.0</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>1.2 =&gt; 4.6</td>
<td>1.1 =&gt; 4.9</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>9.5 =&gt; 16.7</td>
<td>2.5 =&gt; 4.5</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>7.9 =&gt; 10.2</td>
<td>2.5 =&gt; 4.7</td>
</tr>
</tbody>
</table>

The longest I/O wait time is not dependent on the number of pipe nodes: npipe. The memory size of the agent is a key factor for performance above 4 pipe nodes.

The effect of the agent memory size:

By comparing the times in columns 1 and 2, it is obvious that above 4 pipe nodes the agent memory size is a key factor of performance. I/O times can be reduced by factors 2 to 4 when the minimum memory size of the agent is taken “large enough”.

The memory management of the agent process can be controlled by 3 environment variables (see [2] for details): MPP_AGENT_IO_MEM_MIN, _MAX and _INC. The standard values of these are: min = 512 kw max = 8 Mw and inc = 256 kw. By starting with a minimum size of 16 or more Mwords, and giving an increment of about 4 Mw, allows the agent to concentrate on its functions instead of spending some time to increase its stack size when dealing with a large amount of system calls.

The same effect on I/O performance will be also found on table (2).

Effect of the number of pipe nodes on the I/O wait time:

On table (1) values indicate a very small influence of the number of pipe nodes on the maximum I/O wait time across all PEs. This reflects the fact that it takes about the same time to read a given file, either with just one PE reading all records, or several pipe nodes reading a subset of them: the longest time is to bring the data from the disks into the buffers of the front end computer. The subsequent distribution across all PEs is relatively constant since the total number of PEs is constant. In this configuration an optimal number of pipe nodes could be taken from 1 to 4 for example.

Scalability of the I/O wait time with the amount of data:

How the I/O wait time scales with the amount of data flowing through a fixed number of pipe nodes can be seen on table (2): with 8 pipes, (and a “large enough” agent memory) the I/O time scales just like the number of records to read. This has been found also for 1, 2, 4 or 16 pipes. This again reflects the real time taken to bring the data from disks to the I/O buffers, so the largest the file, the longer it takes to read it, independently of the number of pipes. This again calls for a small number of pipe nodes when reading only one file.

Table 2

<table>
<thead>
<tr>
<th>NPIPE</th>
<th>RECORDS PER PIPE NODE</th>
<th>T: STANDARD AGENT</th>
<th>T: AGENT MEMORY SIZE = 30 Mw</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2</td>
<td>0.4 =&gt; 0.8</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>9.5 =&gt; 16.7</td>
<td>2.5 =&gt; 4.5, 1.5 =&gt; 3.6</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>5.0 =&gt; 11.7</td>
<td>1.9 =&gt; 9.2</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
<td>4.3 =&gt; 21.3</td>
<td></td>
</tr>
</tbody>
</table>

Writing different (private) files with sequential access:

In the second experiment we want to see how the I/O wait time scales with both the amount of data and the number of pipe nodes in a write operation. Files must be different for different nodes, and the easiest way is to write them sequentially. We concentrate here on the I/O time measured on the pipe nodes. Since writing in another PE with shmem-put or PVM is a non blocking operation, the non-pipe nodes are not concerned by the I/O time (excepted at some synchronization points with the pipe nodes). On table (3) the extreme values of the I/O wait time are reported. They include:

\[
t_1 = \text{irtc}(())
\]

open file for writing
do i = 1, N$PES/npipe
    write one record of 100 000 words
enddo
close the file
\[
t_2 = \text{irtc}()
\]

print *, t2 - t1

Times are in seconds and the agent memory size is standard. The global amount of data written is then 100 000 * N$PES words.

Table (3) shows again that the longest I/O time scales like the number of records to write, but depends very slightly of the number of pipes when 2 pipes or more are involved. However the scaling between 1 and 2 pipes indicate a real decrease of the I/O time. The weak dependence of the I/O time with the number of files written favours again a small number of pipe nodes. The optimum value depends also on the disk configuration and the file system partitioning.
**Table 3**

<table>
<thead>
<tr>
<th>NPIPE</th>
<th>N$\text{#PES}=32$</th>
<th>N$\text{#PES}=64$</th>
<th>N$\text{#PES}=128$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.4</td>
<td>9.1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.6</td>
<td>4.3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.8 $\Rightarrow$ 2.9</td>
<td>3.7 $\Rightarrow$ 3.8</td>
<td>6.3 $\Rightarrow$ 10.1</td>
</tr>
<tr>
<td>8</td>
<td>1.1 $\Rightarrow$ 2.8</td>
<td>3.4 $\Rightarrow$ 3.9</td>
<td>7.6 $\Rightarrow$ 9.6</td>
</tr>
<tr>
<td>16</td>
<td>2.8 $\Rightarrow$ 3.1</td>
<td>2.7 $\Rightarrow$ 7.8</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>7.8 $\Rightarrow$ 9.4</td>
<td></td>
</tr>
</tbody>
</table>

**Single Record and Global I/O Wait Times**

In tables (4) and (5) we report more detailed measurements of individual record I/O wait time altogether with a measurement of the global I/O wait time: for all pipe nodes. Global timings, for a given amount of data, are measured by:

```fortran
call barrier()
1 = itrc()
I/O activity: open + R/W + close
2 = itrc()
tg = t2 - t1
print *, tg
```

tg values are private to each pipe node, but since the real times are taken after a barrier, the actual values are almost identical across all nodes.

**Table 4**

<table>
<thead>
<tr>
<th>NPIPE</th>
<th>NREC</th>
<th>WRITE</th>
<th>READ</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR</td>
<td>TG</td>
<td>TR</td>
<td>TG</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>0.1 $\Rightarrow$ 0.15</td>
<td>11.2</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>0.1 $\Rightarrow$ 0.19</td>
<td>6.86</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>0.1 $\Rightarrow$ 0.2</td>
<td>5.47</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0.164 $\Rightarrow$ 0.2</td>
<td>8.47</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>0.164</td>
<td>10.29</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>0.164</td>
<td>9.47</td>
</tr>
</tbody>
</table>

Experiments in table (4) are carried with Fortran DIRECT ACCESS files (one per pipe node). All records are 100 000 words long, and the agent memory size is the standard one. 1r refers to I/O wait time to read or write one individual record. tg refers to the global I/O time as defined above. All times are in seconds. The total amount of data is 64 $\times$ 100 000 words or 51.2 Mbytes. The number of pipe nodes has only a weak effect on the global I/O wait times, with a tendency to reduce the time when the number of pipe nodes increases. Above 8 pipes nodes the reduction is not very significant, and it should be measured with a larger agent memory size. Some negative effects of the standard size of the agent can be seen here (mainly on the writes).

Individual record I/O times, when followed record by record, show the effects of the buffers: most times are close to the minimum, but a few are close to the maximum: this occurs when the buffers need to be flushed. Re-use of data already in buffers can thus greatly enhance the overall performance. This favours in some applications the use of Direct Access with a buffer large enough to contain several records.

**Conclusions**

From these experiments using “pipe nodes” with FORTRAN Private I/O we observe some simple rules:

i) Proportional to the total amount of data, but

ii) Only weakly dependent on the number of pipe nodes. For applications that are not I/O bound, it is thus simpler to use a few nodes (typically 1 to 8) to perform I/O and redistribute the data through the T3D network to other PEs. Choosing FORTRAN direct Access has some benefits when reading one or several private files. Another advantage of Direct Access is that the files contains only pure data (no control word). A disadvantage is that the conversion between IEEE and CRAY format cannot be done on line. With sequential Access the performances are close to those with Direct Access (even better for writes), and on line conversion to CRAY format can be done with the assign command.

The I/O performances can decrease rapidly if the rate of system calls per second increases and the agent memory size is too small to contain all the relevant tables and buffers.

This problem can easily be solved by imposing a large memory side and a large increment with the MPP_AGENT_IO_MEM ... environment variables.

Distributing data from the pipe nodes to elementary PEs can be done most efficiently by non-blocking transfers with shmem put library calls. When a simple broadcast is needed, shmem-broadcast should be used since all PEs are co-operating for the broadcast, ensuring a log (NSPES) scaling of the global broadcast time. More performance data on these functions can be found in [3].

If portability of the code is a key factor, PVM functions give also good results. The effect of the PVM transfers on the global performance can be seen by comparing results of table (1) or (2) with those of table (4): in table (1) 25.6 Mbytes are transferred in about 5 sec. Thus at an average rate of 5.1 MB/sec, this includes the PVM transfer time. From table (4), with npipe = 1, the transfert of 51.2 Mbyte is done in about 10 sec on the pipe node, this at a comparable rate. PVM transfers to other nodes have little effect on the overall I/O wait time. Table (4) also shows that, by reading different files on several pipe nodes can improve the overall performance by introducing a true parallelism of the I/O.

**Bibliography**


Polymeric materials pose a challenge for Monte Carlo simulations, because of widely spread length and time scales involved. These range from the vibration of chemical bonds with time scales of picoseconds to macroscopic phase separation, which takes place on a time scale of days. Therefore only coarse grained lattice models, which do not treat the smallest atomic length scales explicitly, render simulations feasible.

While the athermal version can be vectorized rather efficiently, in the presence of thermal interactions, as required for this study, the simulation of large polymer systems is beyond the capabilities of vector processors. Therefore MPP systems with high performance PEs are a promising alternative, additionally high bandwidth and low latency interconnects are required.

We present the algorithm in the framework of the bond fluctuation model and employ a two dimensional geometric decomposition scheme. The implementation and optimization on the T3D are discussed. The program scales well with the number of processors and less than 4 PE achieve the performance as a vectorizing program on a single Y-MP processor.
Within the framework of a Monte Carlo simulation it is necessary to generate a large number of statistically independent system configurations. Therefore, the algorithm lets the polymer conformations evolve via random monomer displacements, which are illustrated in figure 1. Since each movement displaces the center of mass of a polymeric coil, consisting of N monomers, only by a distance of the order 1/N, one needs a large number of attempted movements (typically 10000 Monte Carlo steps, 1 MCS = attempt to move each monomer once) between two statistically independent configurations. The random monomer displacements are accepted by a scheme which generates a stochastic sequence of configurations with the correct statistical weight. Suppose a randomly chosen monomer (black square) attempts to hop one lattice unit in a randomly chosen direction. The new position (thick framed square) is accepted or rejected according to the following sequential procedure:

1. If the new bond vectors (dashed lines) belong to the allowed model bonds, proceed further.
2. If the new monomer position does not violate the excluded volume restriction, i.e. if the grid points in the light shaded region are not occupied by other monomers, proceed further.
3. Calculate the energy change \( \Delta E \) associated with the displacement.
4. Finally accept the move with standard Metropolis rate, i.e. if \( e^{-\beta \Delta E} \) is greater than a random number uniformly distributed between 0 and 1.

If the move is accepted, the occupancy of the old lattice site is removed, the information is transferred to the new grid point, and the bonding vectors of the preceding and succeeding monomers are updated. Consequently a monomer movement requires only local information, which is at most 3 lattice units apart in each direction. This defines the interference range of the monomer movement, which is the lightly shaded area in figure 1.

**Implementation on the T3D**

Since there are no long range interactions, we decided to exploit geometrical parallelism. The general features of geometric parallelization of lattice models (e.g. the Ising model) are well known [7]. However there are some aspects specific to the BFM:

- The degrees of freedom are related to monomers and not to states of the lattice sites. These degrees are dilutely distributed with an average density of 1/16. Only about 15% of the attempted monomer movements are accepted, because of the excluded volume and thermal constraints.
- The CPU-time, required for an attempted monomer hop, may vary substantially from attempt to attempt, because of the various constraints to be checked for updating the lattice should the move be accepted. Additionally the monomers can cross the logical PE boundaries during their motion, resulting in a slight time dependence of the number of particles on each PE.

Both effects lead to a fluctuation of the computational load of each PE.

The system with geometry LxDxL is partitioned into nxm columns of size \((L/n)xDx(L/m)\) (figure 2). Each column is placed in the local memory of a PE. PEs work locally on their respective column, but in addition require information about the adjacent 3 lattice slices in the x and y direction. The width of this boundary region, which is marked by the shaded area in figure 2, is set by the interference range of a trial movement. Note that the boundary region is not small compared to the column size. Each column is split further into 4 subcolumns, in order to prevent the overlap of interference ranges between different monomers that are being moved simultaneously. Monomers of the same subcolumn on each PE are moved simultaneously.

If a movement is accepted, the old and new monomer positions and the grid points of the adjacent monomers along the chain are updated and, if the sites belong to the boundary region, the pertinent information is stored in a list. The PEs work periodically on their monomer lists. After processing the monomers in one subcolumn in a random order, the changes in the boundary region are communicated to the three neighbours, as indicated in figure 2. Then each PE generates random numbers and updates its boundary in turn.

**Program Structure**

The code is implemented in ANSI-C and was first functionally tested on a workstation cluster, with communication performed via PVM. Practically useful simulations however are not feasible on a workstation cluster, because communication is too slow, and response times are unpredictable, thus inhibiting scalability to a large number of processors. We therefore ported the program to the massively parallel processor system T3D [8].

A complete simulation consists basically of first generating an initial start configuration (equilibration of an athermal melt)
using a separate program, and then performing the following steps:

- reading start configuration from disk, required by each PE
- perform the simulation for a predefined number of iterations
- save the resulting configuration, i.e. each PE writes its partial result
- transform the resulting configuration to a subsequent start configuration.

**Startup**

PE 0 reads the initial configuration and then broadcasts the values to all PEs. The broadcast is implemented as a binary tree, thus LDn steps are required to send the data to all participating n PEs. Measurements indicate an effective transfer rate of more than 150 MB/sec from PE 0, regardless of the number of PEs and including all call overheads. Thus for example a 256 PE system will, in the final step, have 128 PEs sending data, each at 150 MB/sec, i.e. an aggregate transfer rate of almost 20 GB/sec.

**Communication**

The fastest high-level communication mechanisms are the so-called shared memory routines (e.g. shmem_put) [9]. Their main advantage compared to PVM is the capability to directly address another PEs memory space along with a startup time of less than 1 msec, which is about a factor of 5 lower than PVM.

From a programming point of view, the simplest and most elegant approach is for each PE to perform its local computations, then move (shmem_put operation) the update information into the neighbours receive area, and wait on a global barrier. Once every PE has arrived at the barrier, each PE may continue without additional checking or acknowledging.

As pointed out before, there are variations in the amount of work performed by each PE within a given MC step, thus leading to local load imbalance, although over time each PE will perform the same amount of work. So we implemented local synchronization by handshaking between the neighbours, resulting in a reduction of communication time of typically 50%. The basic effect between global and local synchronization for an imbalanced load is shown in figure 3, i.e. a PE may immediately proceed after having received the update information from its neighbours.

**Results**

![Figure 3 – Effect of global vs. local synchronization](image)

**Figure 4 – Performance in terms of updates/sec and CRAY Y-MP CPU equivalence**

The program described above achieves about 140,000 attempted movements per second per PE, i.e. less than 4 PEs are about as efficient as a highly vectorized version of a similar code on one CRAY Y-MP processor. The very good scalability of this code enables us to investigate polymeric systems, that would be impossible using previously available computers. In figure 4 we have summarized the measurements in terms of updates per second and CRAY Y-MP CPU equivalence for the 512x64x512 system, which contains already more than a million of monomers and is the largest model of a polymer blend simulated thus far.

Investing 15,000 PE-hours, we investigated structural and thermodynamical properties of interfaces in these complex systems [10].

**Acknowledgements**

It is a great pleasure to thank K. Binder for many stimulating discussions and encouragement and W. Oed for invaluable help in porting and optimizing the program for the T3D. Partial support from the DFG under grant Bi314/3 and generous access to the T3D at CINECA, EPFL and EPCC are also acknowledged.

**References**

Evolutionary Algorithms (EAs) are a class of artificial adaptive processes that find their origin and inspiration in the biological natural selection mechanisms. Genetic algorithms (GAs) [11] seek optimal or near-optimal solutions to hard search and learning problems by giving more chances of survival to fitter individuals in an evolving population in which each individual represents a feasible solution to the given problem through a suitably coded string of symbols. New solutions are explored by mixing good individuals and artificial mutations are used to prevent premature convergence to local optima by randomly sampling new points in the search space. Evolutionary algorithms have found increasing application to many problems in diverse areas such as hard function and combinatorial optimization, neural nets evolution, routing, planning and scheduling, management and economics, machine learning and robotics and pattern recognition.

Genetic programming (GP) is a variation of genetic algorithms in which the evolving individuals are themselves computer programs instead of fixed length strings from a limited alphabet of symbols [6]. Programs are represented as trees with ordered branches in which the internal nodes are functions and the leaves are the so-called terminals of the problem. The search space in genetic programming is the space of all computer programs composed of functions and terminals appropriate to the problem domain. Suitable functions and terminals are determined for the problem at hand and an initial random population of trees is constructed. The population then evolves with fitness being associated to the actual execution of the program and with genetic operators adapted to the tree representation. The crossover operation first selects a random crossover point in each parent tree and then exchanges the sub-trees, giving rise to two offspring trees. Examples related to our application will be given in the next section. There are also provisions for preventing trees from becoming too deep, for simplifying trees and for compressing trees that perform a useful functions into a single reusable module.

Genetic programming is well-suited to parallel implementation. The most popular parallel models are the fine-grained or grid models, and the coarse-grain or island models. In the grid models, large populations of individuals are spatially distributed on a low-dimensional grid and individuals interact locally within a small neighborhood. In the island model the population is subdivided into smaller subpopulations which evolve independently and simultaneously according to a standard EA. Periodic migrations of some selected individuals between islands allow to inject new diversity into converging sub-populations. Micro-processor-based distributed memory machines and workstation clusters are well adapted for the implementation of this model. The advantage of parallel EAs for difficult problems is that they can handle larger populations in reasonable times and favor cooperativity in the search for good solutions ([5], [15]). In the following section we introduce the binary decision diagrams optimization problem and explain in more detail the parallel GP implementation used to solve it.
**GENETIC PROGRAMMING FOR BINARY DECISION DIAGRAMS OPTIMIZATION**

**BINARY DECISION DIAGRAMS**

A binary decision diagram (BDD) is a type of oriented graph used notably for the description of algorithms. It assembles, according to some rules, two types of nodes: the decision or test node and the output node. The decision node is equivalent to an if-then-else instruction: it realizes a test on a binary variable and, according to this value, indicates the node following. The output node produces a value. The two rules of assembly are: there is one and only one initial node (the entry point of the algorithm); the output point of a node can be connected to only one entry point of another node.

A binary decision tree is a binary decision diagram that respects a third rule of assembly: any entry point of a node is connected to only one preceding node.

Since [8] it has been demonstrated that all logical boolean function can be represented by a binary decision diagram. This type of representation finds applications in the test and the implementation of logical functions [3]: the function of a decision node can be implemented by a multiplexor or demultiplexor circuit and a binary decision diagram can be implemented by an interconnection of these circuits. In all these cases, the minimalisation of the number of nodes used is important, for the cost and/or the time of execution of the function. Nevertheless, the complexity of this minimalisation is such that in most cases approximate solutions are accepted [12].

A renewal of interest on the minimalisation of binary decision diagrams is born with the appearance on the market of programmable circuits named FPGA (Field-Programmable Gate Array) [13] [10]. These circuits appear under the form of an array of identical cells (the logic cells), where the user can program the function inside every logic cell (among some possible) and interconnections between cells. Each FPGA manufacturer proposes a different type of logical cells and interconnections. Some, as Actel [1], propose very simple cells, formed of a simple multiplexor circuit. A minimal binary decision diagram can therefore drive to an optimal utilization of the FPGA cells.

**THE GENETIC PROGRAMMING REPRESENTATION**

In preparing to use genetic programming to solve a problem one has to decide on the set of terminals, the set of primitive functions, the fitness measure, the stopping criterion and the values of some parameters such as population size and crossover rate [6]. The fitness of an individual is defined in our case to be the difference between a perfect solution and the actual number of hits of the given individual on all the input combinations of values. Therefore, a fitness value of 0 means that the individual correctly solves the problem.

While the choice is seldom unique, the nature of the problem suggests suitable terminals and functions. For the representation of binary decision diagrams we tried two representations. In the first one the terminal set was made by the operation codes and the output codes. The only function operating on those terminals was the three-branches IF function. Random-constructed trees are not guaranteed to be valid decision diagrams for our problem since the IF function itself can be the first argument (i.e. the test condition) of another IF function. Furthermore, there is nothing to prevent an output code from also being the first argument to the IF function. Finally, input variables can only appear as test conditions. Likewise, crossing-over valid trees will not necessarily yield admissible offspring (see fig. 1). We therefore penalized invalid trees by giving to them the worst fitness value in order for them to be less likely to be selected for reproduction.

In the second solution we avoided mixing terminals by creating a specialized IF function for each operation code. Example diagrams and a crossover operation are shown for a simple case in fig. 2. With the latter choice of functions and terminals all trees are guaranteed to be valid. Results of the runs of parallel genetic programming using both representations will be discussed in section 3. We now briefly describe our parallel GP implementation choices.

**THE PARALLEL IMPLEMENTATION**

The parallel architectures that best matches the rather coarse grain and variable length of genetic programs are micro-processor-based distributed memory machines, including workstation clusters. On these machines it is easy to implement the island model. Good results were obtained in [9] and [7] using a similar computing architecture.
The T3D multicomputer [4] is a DEC/Alpha-based MIMD machine. The processors are connected by a fast bidirectional 3-D torus interconnect network. The memory of the machine is physically distributed although, depending on the programming model used, it can be globally addressable. Communication latency is low and bandwidth is high due to latency hiding and data transfer optimized hardware and easy routing mechanisms. The T3D array is connected through I/O nodes to a Cray Y-MP host machine on which all program development takes place. Access to peripherals such as disks, tapes and the network is through the host. Three different programming models are available on the T3D: message-passing, work and data sharing using a global address space (CRAFT) and data parallelism.

The message-passing approach perfectly suits the island model for parallel genetic programming. It is based on PVM which is a standard message-passing environment.

We started from the publicly available sgpc GP program [14]. The PVM-based code parallelization was easy except perhaps for the modifications needed to pack and unpack program trees to be sent to other subpopulations (islands). This requires linearizing the trees to be packed in a message buffer and rebuilding them at the destination subpopulation in the new processor’s private address space. For efficiency reasons, we pack all migrating individuals in a single message, which minimizes message startup and transmission time.

After code parallelization, suitable values for a number of parameters of the distributed algorithm must be chosen. Besides the usual global population GP values for each subpopulation one has to define the topology and the number of the communicating subpopulations, the size \( N \) of the subpopulations, the migration frequency \( M \), the number of migrating individuals \( K \) and the individual replacement policy. We found suitable values by trial and error, running many times the parallel algorithm on the well-understood multiplexor problem [6]. The size of the subpopulations for the present problem was thus set at 400, migration took place every 7 generations and the number of individuals exchanged was 8 to 10% of the subpopulation size. These values were close to those used in [9] and in [7].

We experimented with only one processor topology: the ring, and we tested two synchronous exchange policies: simply passing individuals to the next island, alternating directions at each swap, and passing individuals «modulo» the swap number (fig. 3). The «modulo» swap gave the best results and we retained it for the subsequent tests. We choose to migrate the best \( K \) individuals from each island.

The replacement policy used was that the new \( K \) individuals displace the worst \( K \) individuals of the receiving population. Here also other alternatives are possible. A GP run is terminated either by finding a solution (i.e., a 0 fitness individual) in any subpopulation or by reaching a maximum number of generations. The following pseudo-code gives a schematic description of the algorithm:

```plaintext
initialize \( P \) subpopulations of size \( N \) each
while termination condition not met do
  for each subpopulation do in parallel
    evaluate and select individuals by fitness
end while
```

### Results and Conclusions

We did many parallel GP runs for each of the two choices of functions and terminals described in the previous section. The parallel GP based on the first terminal and function set never found a solution in the allowed maximum number of generations (60). A typical run is shown in Fig. 4, in which the number of individuals having a given fitness value generation by generation is presented. It is clear that there is little improvement after about 40 generations and the search stagnates. The best-of-all-runs individual attained a fitness value of 2. The sequential program was never able to find a fitness value better than 4 even when given a maximum number of 100 generations. Clearly, the reason for the unsatisfactory performance both of the sequential and parallel algorithms was the presence of a large percentage of invalid diagrams in the populations. This is so in spite of the worst possible fitness value given to them because crossover, being unrestricted, continuously produces invalid diagrams.

The second terminal and function set choice proved to be much more adequate. The parallel GP was able to find the correct solution on most of the 30 runs. A typical successful run is graphically depicted in fig. 5. Comparing it with fig. 4, it is seen that the average fitness is much lower and that the run quickly converges to a 0 fitness solution.

Not only did the parallel algorithm perform much better from the point of view of computing times, which was obviously expected, it also converged more often to the correct solution than the sequential one using a smaller number of fitness evaluations i.e., with a reduced computational effort. For instance, a particular run on a
8-processor system took 33 seconds to complete successfully with a population of 400 individuals per processor whereas for the same total population size (i.e., 400 x 8 = 3200) a sequential execution on one processor took approximately 10 minutes to complete. We observed the same effects in parallel GA systems [9] and the detailed results reported in [7] also agree with this general trend.

**REFERENCES**


---

**A PARALLEL DIVIDE AND CONQUER EIGENSOLVER**

*by Christopher P. Potter, PATP project, EPFL*

Cet article présente l’implémentation de l’algorithme “Divide & Conquer” sur un ordinateur à mémoire distribuée/partagée, le T3D. L’algorithme “Divide & Conquer” permet de résoudre des problèmes de valeurs propres plus rapidement que l’usuel QR, et plus important, est plus parallélisable. L’inconvénient de cette méthode est une certaine instabilité des vecteurs propres pour de rares matrices, mais des solutions ont été récemment trouvées.

This paper describes the implementation of the Divide and Conquer algorithm on a distributed/shared memory parallel computer: the T3D. The Divide and Conquer algorithm allows the solution of eigenproblems in a faster way than the usual QR and, more importantly, is easily suitable for parallelization. A drawback of this method is some orthogonality problems for some pathological cases, but solutions have recently been found.
INTRODUCTION

The project started with the need for a fast eigenvalue solver for medium sized matrices (n ≈ 500) for the EPFL parallel computer (a 256 node T3D). It appeared from [3] that an obvious algorithm for parallel computers would be the Cuppen’s Divide and Conquer algorithm [1]. This algorithm started to be popular when it was proved that even the serial implementation could be much faster than the common QR in some cases e.g. clustered eigenvalues [4].

A major drawback of this algorithm was its instability in a few pathological cases where the orthogonality of the eigenvectors was not achieved. Now two techniques can be applied to reduce this side effect, the first one using extended precision [7], and the second by a correction stage to compute the eigenvectors [6].

DIVIDE AND CONQUER METHOD

RANK-ONE MODIFICATION

The Divide & Conquer method is based on reference [1] which used the rank-one modification algorithm found in [8].

Consider a matrix T, real symmetric and tridiagonal of size n x n:

\[
T = \begin{bmatrix}
  t_{11} & t_{1,2} & \cdots & t_{1,n-1} \\
  t_{1,2} & t_{22} & \cdots & t_{2,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  t_{1,n-1} & t_{2,n-1} & \cdots & t_{n-1,n-1} \\
\end{bmatrix}
\]

We define two sub-matrices, T1 being the upper-left one with the k'th diagonal element modified, and T2 the one with the (k+1)'th diagonal element modified.

Our matrix T can now be rewritten as the sum of two matrices:

\[
T = T_1 + T_2
\]

The problem, which is commonly known as the rank-one modification problem, consists now to diagonalize \( T \), where \( T_{11} \) and \( T_{22} \) are diagonal and \( T_{12} = T_{21} = 0 \):

\[
T = \begin{bmatrix}
  T_{11} & T_{12} \\
  T_{21} & T_{22} \\
\end{bmatrix}
\]

The eigenvector corresponding to eigenvalue \( \lambda_i \) is:

\[
\mathbf{v}_i = \begin{bmatrix}
  v_{i1} \\
  v_{i2} \\
  \vdots \\
  v_{in} \\
\end{bmatrix}
\]

We assume that \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \) are known and find \( \mathbf{v}_3 \) by solving the secular equation:

\[
f(\lambda) = 1 + \sum_{j=1}^{n-1} \frac{\lambda_j - \lambda}{\lambda_j - \lambda_i}
\]

Finding the eigenvalues of \( D + r \mathbf{v}_i \mathbf{v}_i^T \) can be done by solving a specific equation (the secular equation) and the eigenvectors by multiplying \( Q \) by the eigenvectors of \( D + r \mathbf{v}_i \mathbf{v}_i^T \).

THE SECULAR EQUATION

It can be shown [9] that the eigenvalues \( \bar{\lambda}_i \) of \( D + r \mathbf{v}_i \mathbf{v}_i^T \) can be found by finding the roots of the secular equation:

\[
f(\lambda) = 1 + \sum_{i=1}^{n} \frac{\lambda_i - \lambda}{\lambda_i - \bar{\lambda}_i}
\]

The eigenvectors

The eigenvector \( \mathbf{v}_i \) corresponding to eigenvalue \( \bar{\lambda}_i \) is:

\[
\mathbf{v}_i = \frac{\mathbf{v}_i}{\|D - \bar{\lambda}_i I\|_2}
\]

where

\[
\mathbf{v}_i = \mathbf{v}_i - \sum_{j=1}^{i-1} \mathbf{v}_i \mathbf{v}_j^T \mathbf{v}_j
\]

So we can rewrite (1) as:

\[
\mathbf{v}_i = \begin{bmatrix}
  z_1 \\
  z_2 \\
  \vdots \\
  z_n \\
\end{bmatrix}
\]

figure 1 — The secular equation

\[
\bar{\lambda}_i = \frac{z_1}{d_1 - 1}, \ldots, \frac{z_n}{d_n - 1}
\]

\[
\sqrt{\sum_{i=1}^{n} (\bar{\lambda}_i - 1)^2}
\]
Once these eigenvectors are determined, the one for $T$ can be calculated by: $x_i = Qz_i$

**PRECISION OF THE EIGENVECTOR**

In some cases (i.e., $\frac{1}{d_i}$ is small but not enough to deflate), the ratio $\frac{1}{d_i - 1}$ cannot be computed with sufficient precision. It results in a wrong eigenvalue, as well as a non-orthogonal eigenvector matrix. It should be underlined that due to the fact that the Divide-and-Conquer algorithm uses the eigenvectors of the previous stage to compute the eigenvalues of the new stage, this lack of precision can induce a disastrous error accumulation.

Fortunately, several techniques can prevent such effects. The first one [7] is to use double precision (or simulated double precision) to compute the secular equation (3). The second one [6] is to compute $\tilde{z}_i$, a modification of $z_i$ taking into account all newly computed eigenvalues and all last iteration eigenvalues $d_i$. With this $\tilde{z}_i$, all eigenvectors can be computed to high accuracy.

**DEFLATION**

When some eigenvalues (the $d_i$’s) are equal or if one of the elements $z_i$ is zero, then a deflation procedure [8] must be used to satisfy (4). Note that the above two statements should be replaced in finite precision arithmetic by nearly equal and nearly zero. The tolerance factor must be chosen carefully to avoid loss of precision [1] [8].

**In practice**

- If $d_i = 0$, then $G = 0$, and the $z_i$ is computed by $D + rz_i$. This is done recursively unless $d_i = 0$, so due to the next affirmation, $z_i$ will not have to be computed.
- If $z_i = 0$ for some $i$, then $d_i$, so the corresponding eigenvalue does not have to be calculated.

This deflation process explains why the divide-and-conquer method is so fast. Because for most matrices, the inherent structure allows large deflation and therefore saves huge amounts of computer time (between 2 to 80 time faster).

**PARALLEL IMPLEMENTATION**

**GENERALITIES**

The last paragraphs showed how to solve the eigenproblem of $T$ from two sub-matrices $T_1$ and $T_2$ of size $n/2 \times n/2$ (in fact, in the general case, the size of $T_1$ and $T_2$ could be different). This decomposition of $T$ can be done recursively unless we get $k$ submatrices at the leaves of a binary tree. Again, in our case, all submatrices have the same size, so $\frac{1}{2}k \times \frac{1}{2}k$. To simplify the problem on a distributed memory machine (hence avoid very complicated task handling and introduce unbalance), is chosen to be the number of available processors, so every processor will have a small matrix to solve via QR at the root of the tree (see Figure 2).

This decomposition is of course well suited to parallelization because every node (composed of 2 submatrices) at the leaves of the tree can be computed in parallel using a classic algorithm like QR. After this initial step, a certain partition of processors can be used to solve the secular equation independently from the other nodes at the same level of the tree. This partition is composed by all processors at the root of the tree, half of the processors at a level below, etc.

A synchronization point and communication stage (eigenvalues and eigenvectors) is required after the computation of the left and right child of the a node. One should note that as we go from the leave of the tree to the root, the amount of communication per node is doubling for each upper level and the number of eigenvalues to be computed per PE is constant if no deflation occurs.

![Figure 2 — The Divide & Conquer Tree](image)

**LOAD BALANCING ISSUES**

When there is no deflation, the amount of computation for each node of the tree is nearly equal (because the number of iteration to solve the secular equation is not totally constant). But this situation with no deflation is very rare even never encountered by the author.

So, in real cases, the number of deflated eigenvalues can be different on different parts of the tree. This introduce different computational time for each branch of the tree. Since we have a synchronization point when merging two nodes from one level to the upper one, a load imbalance is created. However, as most of the work is done at the root of the tree (where the matrix-matrix multiplication takes a long time), it can be shown [5] that the minimal efficiency will be 85% in the most unbalanced situation.

One should note that in each node of the tree the work is nearly equal because the program distributes the number of non-deflated eigenvalues to the processing processors. The maximum load imbalance in the same partition is one eigenvalue to compute.

**$Q$ and $\bar{Q}$ MATRIX DATA LAYOUT**

When having a close look at the algorithm it can be easily shown that the most computational intensive part is the matrix-matrix multiplication $X = Q\bar{Q}^T$.

The most effective distribution for matrix-matrix multiplication is block-block [3]. However, $\bar{Q}$ is naturally distributed by columns. The reason for this is because every processor, after computing an eigenvalue, (again by solving a part of secular equation (3)) can compute directly the corresponding eigenvector $\tilde{d}_i$ (the $z_i$’s and $d_i$’s needed are local). If this column distribution is not preferred, column
parts of \( \mathbf{Q} \) must be redistributed among processors and Cannon’s algorithm \cite{Cannon} can be used to do the multiplication \( \mathbf{Q} \in \mathbb{R}^{n \times p} \times \mathbb{R}^{n \times p} \) communications per processor).

Considering this column distribution for \( \mathbf{Q} \), the best distribution for \( \mathbf{Q} \) is by row so the matrix-matrix multiplication can be done in place. In this case, there is \( n^2 \) communications per processor.

However, as our implementation was focussed on small matrices, we chose to keep a copy of \( \mathbf{Q} \) on every processor of the same partition so the matrix-matrix multiplication does not require any communications. The major drawback is of course large memory requirement \( an \times n \) matrix must be stored on each PE.

After the completion of each stage, all the eigenvectors/eigenvalues of the left and \( (2n^2 \{2p-1\} \text{and } n(2p-1)) \) processor’s partition of the next level.

\[ \text{RESULTS}
\]

**The parallel algorithm**

Here is a very simplified version of algorithm of what I implemented on the T3D:

- Divide \( \mathbf{T} \) in \( k \) submatrices \( \mathbf{T} \)
- Compute \( \mathbf{T} \) (size \( n/k \times n/k \)) with QR
- Send the eigenvectors/values to all PEs participating at the next level
- New level:
  - Form the secular equation
  - Deflate if possible
  - Dispatch an equal number of eigenvalues to be computed among the participating PEs
  - Compute the eigenvalues \( \mathbf{X} \)
  - Compute the eigenvectors of \( \mathbf{T} \) by doing \( \mathbf{X} = \mathbf{Q} \mathbf{X} \)
  - Send the deflated eigenvectors/values to all PEs participating at the next level
  - New matrix size = 2 x matrix size

\[ \text{enddo} \]

\[ \text{RESULTS and performance} \]

Different matrices have been tested to try to simulate different deflation schemes, orthogonality issues, etc.

We will present here 4 test matrices.

- \( \mathbf{x} \mathbf{x} \mathbf{x} \): A random tridiagonal matrix to simulate a random dense matrix \((-1,1)\) after tridiagonal reduction
- \( \mathbf{b} \mathbf{W} \mathbf{b} \): A Wilkinson tridiagonal matrix defined by
  \[ \mathbf{W} = \begin{pmatrix} 0.1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \]
  \[ \mathbf{T} = \begin{pmatrix} 4 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & 4 \end{pmatrix} \]
- \( \mathbf{1} \mathbf{2} \mathbf{1} \): A tridiagonal matrix with 2’s on the diagonal and 1’s off-diagonal

\[ \text{Residual, orthogonality and deflation ratio} \]

Let’s define here how we computed the residual and the orthogonality.

\[ \text{Residual: } R = \max_{1 \leq i \leq n} \frac{\| \mathbf{x}_i - \mathbf{e} \|}{\| \mathbf{e} \|} \]

where

\[ \text{Deflation ratio: } \alpha = \frac{\| \mathbf{X}_i \|}{\| \mathbf{X}_i \|} \]

To have an idea about the amount of deflation I define the following ratio only at the last iteration:

\[ \text{def} = \frac{\text{number of non deflated values}}{\text{matrix size}} \]

Obviously, a ratio of 1 means that at the last iteration no deflation occurred, and a ratio of 0 means that the full matrix deflated so no computation was done at the last iteration.

\[ \text{Results} \]

**TriDiagonal Matrices**

The random matrix (figure 3) has little deflation (deflation ratio = 0.87). The scalability is quite good even for small matrices (512 x 512). The orthogonality and residuals are acceptable.

Matrix represented by figure 4 has a medium deflation (deflation ratio = 0.50). The orthogonality is good but decreases substantially when the matrix size increases. However, the scalability is less good than for the random matrix. This phenomenon can be explained by the fact that the more deflation there is the less computation is required but the same amount of communication is done.

**Band Matrix**

The band matrix has very little deflation (deflation ratio = 0.95). The figure 5 and table 1 presented here represent the time to fully compute the eigenvalues and eigenvectors of a full real symmetric matrix. This is accomplish by calling a parallel Householder algorithm to tridiagonalize the matrix and then use Divide & Conquer to diagonalize. After this stage, the permutation matrix \( \mathbf{P} \) computed by the householder algorithm is used to compute the eigenvectors of the original matrix \( \mathbf{X} \): \( \mathbf{X'} = \mathbf{P} \mathbf{X} \)
**A Parallel Divide and Conquer Eigensolver**

**Figure 3 — Random Matrix cpu time**

**Figure 4 — Matrix 121 cpu time**

**Figure 5 — Band Matrix cpu time**

**Future Enhancements**

This eigensolver proved to be well suited for the cases it was originally designed for: small matrices. However, the main drawback is the lack of orthogonality of eigenvectors for some matrices like the Wilkinson one and therefore, stabilization techniques should be implemented as soon as possible. This has not been done already because only the "double precision" techniques do not require extra communication and "double precision" is not available yet on the T3D. Gu's method implies another communication step and has, for this reason, not been selected.

Another problem is the memory usage. The current implementation keeps an on each PE memory and therefore the maximum matrix size solvable with the eigensolver is around 2000 x 2000. If enough people show interest for bigger matrices, matrix could be distributed. This will cause new problems, specially to find the best topology for distributing .

**Acknowledgement**

This project has been carried out in the framework of the EPFL-PATP project (Parallel Applications Technology Program) which is a joined collaboration between the EPFL and CRAY Research Inc., and was performed on the T3D at EPFL.

I would like to thank Michel Roche, Olivier Besson and Andrew Canning from CRAY Research Inc. for their useful help on various topics.

**References**


E-mail: potter@patpserv.epfl.ch

<table>
<thead>
<tr>
<th>PE</th>
<th>n</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>O</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>12/11.1</td>
<td>6.38/5.35</td>
<td>3.35/27.1</td>
<td>1.06/1.45</td>
<td>1.11/0.82</td>
<td>0.71/0.51</td>
<td>0.56/0.36</td>
<td>2.10^3</td>
<td>0.6</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>00.4/20.2</td>
<td>44.2/37.2</td>
<td>23.4/19.4</td>
<td>11.3/0.4</td>
<td>4.6/2.3</td>
<td>2.94/2.94</td>
<td>3.18/1.80</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>NR</td>
<td>NR</td>
<td>164/1137</td>
<td>63.0/0.7</td>
<td>42.9/36.4</td>
<td>24.1/19.4</td>
<td>14.2/11</td>
<td>3.10^4</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

Table 1
Le propos de cet article est de faire le point sur un certain nombre de facteurs qui freinent encore la diffusion et l'avance de l'informatique parallèle. Le succès de l'implantation parallèle d'une application dépend de l'interaction efficace entre plusieurs composantes. Un degré de complexité trop important pour un utilisateur lambda est vite atteint. Cet article suggère qu'un des éléments clé pour le succès serait le choix d'un modèle de programmation largement accepté par la communauté, préalable à une meilleure différenciation des problèmes à résoudre et qui donnerait à l'informatique parallèle l'impulsion nécessaire pour progresser et diffuser.

This article considers some of the issues which are impeding the spread and advance of parallel computing. Currently for the successful implementation of an application on a parallel machine the efficient interaction and coordination of many components are required. This gives rise to a level of complexity which is difficult for an average user to deal with. The article suggests that one of the things that is needed is a widely accepted model of parallel computation. This would enable the separation of concerns and for parallel computing to progress and expanded.

INTRODUCTION

One of the problems with computing is the rate at which developments take place, this means that sometimes ideas are not fully explored or digested before new ideas and innovations are presented. In a real sense this has happened in parallel computing (in this article I use the term parallel computing to cover supercomputing, high performance computing etc.) with consequences for the future development of the subject. This article reviews some of the problems and suggests some solutions - by no means complete solutions. One of the major distinguishing features of sequential computing is that there was/is a single model of computation which is widely accepted, the so called von Neumann model. The advantage of having a single model is that architect designers, application writers, algorithm designers, language designers and programmers are all working on their particular tasks using the same model. Hence there is a certain commonality in the results of their efforts, for example, the programmer is aware of this model and knows, perhaps unconsciously, that the compiler is based on the same model. In the case of parallel computing, supercomputing or high performance computing this is not the case, currently there is no single model of computation for parallel computing. This represents a major impediment to the development of and widespread introduction of parallel computing.

Given that sequential computing has had a model of computation for the last 40 years enables an insight to be gained as to how developments have evolved, how long it has taken, what have been the major problems etc. The development of programming languages gives an interesting insight. High level languages were not the immediate choice of the early programmers, the need to keep the expensive hardware utilised was a dominate consideration, but eventually the necessity of a high level language became accepted such that to day only very specialised tasks are coded in assembly language. After the acceptance of high level languages their evolution took more than a decade before the concept of abstract programming became established. Fortran made the contribution of high level programming, Algol the contribution of block structure, Pascal the contribution of data structures and Simula the contribution of the module. A slow evolution as more experience and understanding was gained and documented - the same pattern of experimentation and development is necessary for parallel computing.

EDUCATION

Computer Science is now a well established subject within universities and computer scientists are reasonably confident that they know how to construct software solutions in the small but not in the large. University courses expounding this approach are based primarily on the sequential computing paradigm - students spend three or four years studying all aspects of sequential computers. Afterwards if they go on to a higher degree or take a job in industry they may study parallel computing, usually it is self taught with no formal instruction. Is this a reasonable scenario for a topic which by all indications is more difficult than sequential computing and perhaps by an order of magnitude? Should the norm be parallel computing? On which the university curricula should be based with sequential computing being mentioned (in passing) as a special case. Or is parallel computing such a specialised technology (as at present) that it should be designated as such because of its apparent complexity etc.?

Some one who is a competent and well trained sequential programmer is expected to learn a whole new series of techniques to program parallel machines, which are in many instances still at the research stage. For example, taking this average programmer and introducing him/her to parallel computing requires an explanation of topics such as synchronisation, granularity, network topology, communication to compute ratio, load balancing, non determinacy, fairness, deadlock, termination, thinking and reasoning in parallel, etc.

Topics like load balancing are currently ‘hot’ research topics, judging by the number of papers in parallel computing conferences and journals but as yet there is no widespread agreement as to the best solution for a massively parallel system.

There are many more topics which could be included in the list
but is it reasonable to expect programmers to deal with these new topics, after all their major concern is to achieve better performance for their application. If they are from a discipline other than computer science why should they invest time and effort in all these topics especially since the track record of parallel computing is one of change - users have no guarantee their effort will be reward or sustained across new generations of parallel machines.

MODEL

What is the solution? The evolution of computer science gives a clue, namely, a separation of concerns. In other words identify what is the legitimate concern of the user and what is the legitimate concern of the system. This, in turn, requires the definition of an abstract machine model so that a clearer perspective is achieved for all parties. Hence the big or 'grand' challenge for parallel computing is to determine a model which has widespread agreement and will form the basis for the future promotion and development of the subject.

Several models have been proposed, for example, the PRAM model [Valiant 1990] and its variants and although restrictive in form it does represent an attempt to provide a model. Perhaps this model is not enough in itself, that it requires several component models, for example,

A MACHINE MODEL
for the lowest level of abstraction - assembly language could be considered as using the machine model;

AN ARCHITECTURAL MODEL
for the interconnection network - it describes how to perform communication;

A COMPUTATIONAL MODEL
to provide an abstract view of a class of computers and reflect the costs and resources of those computers;

A PROGRAMMING MODEL
to enable the description of the computer in terms of the semantics.

Whatever the solution a model for parallel computing is urgently needed to ensure the continued and sustained development of the field.

SOFTWARE

In comparison to sequential computing, parallel computing is like moving from one into three dimensions. The programmers at present must have an awareness of the underlying architecture, software and algorithm design.

In addition, all three components are continually evolving. Twenty five years ago the model of parallel computation wasSIMD in the form of the Illiac IV. It was refined and eventually became a commercial basis for some machines such as the ICL DAP and Thinking Machines’ Connection Machine. Although it offered a limited form of parallelism in that there was a single control unit and many streams of computation operating on different data sets it offered a model which was easy to understand and very suitable for certain types of applications. However, its lack of flexibility was always a source of challenge to hardware designers, software writers etc. and as a result it was soon being overtaken by other developments in hardware parallelism. It is interesting to note that this model of computation is making something of a comeback in the data parallel activities of recent language proposals such as High Performance Fortran.

Vector processor technology started to appear commercially in the seventies and therefore has been available for about two decades. During that time the component technology and price performance ratio has improved and, in addition, the user community has gained an understanding of what is required to program vector processor based machines. This understanding has meant that users know how to structure their loops to obtained efficient execution on vector machines, in addition, there has been some success in automating this process. However, the accumulation of this knowledge and expertise may be redundant in the face of current developments.

Shared memory models were the next fashion as the manufacturers managed to produce machines under this paradigm, however problems of memory contention became severe and limits on the scale of parallelism emerged. At the same time researchers started to gain some understanding of how to program such machines culminating in features such as the monitor [Hoare 1978]. The monitor was an elegant structure proposed in the seventies to provide a structuring technique for the construction of parallel programs. It provided a data structure, the condition and its associated operations to control the interaction of parallel activities. However, no sooner was this being understood and implemented than the hardware was being changed again this time to the distributed memory model. One of the big questions here was how were the machines connected together. Many different interconnections were proposed and implemented such as, linear array, grid, for a time the hypercube was fashionable, now a torus is proving popular.

If all these changes had no effect or a consequence for the user then it would be acceptable but each new change in the architecture does have consequences as the underlying exploitation of the hardware is currently directly related to application efficiency. Hence users are becoming increasing disillusioned or concerned about the amount of time and effort required to move their application onto a new architecture and to achieve a respectable speed up on a changed architecture. They are, perhaps, prepared to do it once - not at the rate of architectural change that is currently happening. The distributed memory model is regarded by many as the more viable model for the future as it is scalable to massively parallel dimensions, however, the programming of such a model is not well understood. The shared memory model is regarded as being easier to program primarily because of the similarities with the sequential programming model but it has technology limitations in the number of processors that can be included. The debate is still open.

LANGUAGES

More specifically, the language developments for parallel computing fall into several distinct categories. Four main approaches have been suggested as a basis for designing imperative programming languages that would promote the wider use of parallel systems [Perrott 1992]; these are:

(1) INVENT A COMPLETELY NEW LANGUAGE

This approach ignores all existing languages and applications in the formulation of the new language. The rationale is that it will facilitate the coherent treatment of all aspects of parallel programming allowing the user to develop and to express a parallel solution.
explicitly. The drawback however is that existing software will have to be re-programmed in the new language which is a labour intensive and error prone procedure. Typical of this approach has been the introduction and adoption of the Ada language.

(2) INTRODUCE FEATURES TO AN EXISTING LANGUAGE WHICH DEAL EXPLICITLY WITH PARALLELISM

This approach should enable existing software to be adapted and transferred to parallel machines, where appropriate, by existing programmers. Many of the extensions have been developed by different groups using the same language base leading to the definition of non-standard variants; this makes the production of a standard for such languages difficult. This approach is reflected in several Fortran proposals recently proposed for distributed memory systems programming, for example, Vienna Fortran [Zima et al. 1992] and Fortran D [Fox et al. 1992], High Performance Fortran [HPF 1993] is an attempt to stop this proliferation.

(3) ENHANCE AN EXISTING SEQUENTIAL LANGUAGE COMPILER TO DETECT PARALLELISM

In this approach the responsibility is placed on the compiler to detect which parts of a sequentially constructed program can be executed in parallel. The principal advantage of this approach is that existing sequential programs can be moved to the target parallel machine exploiting its parallel facilities relatively inexpensively and quickly. This approach has been applied to Fortran where the DO loops of a program are examined to determine if it is possible to spread the iterations of the loop across different processors.

(4) USE A CO-ORDINATION LANGUAGE

This approach integrates a sequential language with a co-ordination language, the latter provides operations that create parallel activities and supports communication. The sequential language allows the programmer to develop single computation activities and the co-ordination language binds these activities together to produce a parallel program. At present this approach has been used with Fortran, Pascal and C and the co-ordination language Linda [Ahuja et al. 1986] and implemented on distributed memory systems such as the Inmos Transputer and Intel iPSC.

This classification excludes all the excellent work that is taking place in functional languages etc. which also offer possibilities for parallel computing. However, the bottom line is that as with hardware there are many different and changing approaches causing problems for the users etc.

NOT ALL GLOOM

In the language arena FORTRAN is still the preferred language for the majority of the users of parallel computers, due mainly to the fact that the majority of users are scientists and engineers and that there is a large amount of software available in FORTRAN. As mentioned earlier one approach to providing a programming language for these machines is to provide a compiler that detects parallelism in sequentially written code, usually FORTRAN. In the case of vector machines this has been a successful approach and has helped with the migration of users onto these machines in the early days of vector machines. In a sense it shielded the users from the reality of programming vector machines.

In the case of parallel machines research has shown that providing an automatic parallelising compiler is a non trivial task and that it may not be possible or only possible in very limited circumstances. For these machines the signs are that the users will have to become more involved in the programming process and have a greater understanding of what is required to program a parallel machine. To this end the process can be evolutionary.

The FORTRAN programming community has produced a new FORTRAN standard, namely, FORTRAN 90 and it is one of the languages that will probably be available on new machines. FORTRAN 90 provides features for declaring parallel data structures and performing operations on such structures. In addition, an ad hoc group of manufacturers, users and compiler writers have produced a data parallel language based on FORTRAN 90, known as High Performance FORTRAN. It provides the user with language features which influence the data layout of a program in the memory of a distributed memory machine; the next evolutionary step would be to introduce features to specify parallel tasks and their interaction. High Performance FORTRAN seems to have widespread support among the parallel computing community and the first compilers are now being produced. Whether High Performance FORTRAN will become the dominant language in the future is still an open question but it is a language that has produced some sort of ‘togetherness’ in the parallel computing community.

At a lower level on parallel machines an international group consisting of vendors and users have produced a message passing interface known as MPI, it defines how processors inside a parallel
machine communicate with one another. It is proposed as an industry wide standard for writing message passing parallel programs and is seen as a replacement for other message passing libraries such as Parmacs etc. Already some manufacturers are producing an implementation of MPI on their parallel computing machines. If MPI becomes accepted then portability between parallel computing machines will be greatly increased, as such MPI is important for the future development of this area.

The benchmarking of parallel computing machines has recently taken a significant step forward with the publication of the Parkbench report [Parkbench, 1994]; this is a method of benchmarking parallel machines. It has been devised by an international group of researchers with expertise in this field. The proposal brings a more scientific approach to the assessment of these machines in a range of areas and should hopefully make it easier to compare the performance characteristics of machines in the future - a promising development.

Over the years a large amount of application software has been developed for vector machines, this is one of the advantages of this type of machine compared to the commodity based machines. In addition, the European Union is sponsoring two major projects in this area, namely, Europort-1 and Europort-2, to port industrial codes onto parallel machines such that they will have a significant impact within a two year time scale. If this is successful it should increase the amount of industrial application software available on parallel machines thus reducing the software availability gap between parallel and vector machines.

It appears that parallel computing is one of the areas in which the peace dividend does not offer benefits as the spending in the defence industries decreases the amount of money available for parallel computing has decreased, some estimates give it as much as a 50% decrease. This has a profound effect on an industry which has seen many casualties and needs a sustainable and expandable customer based to finance the research and development needed to ensure its expansion. However, the primary market has been scientific and engineering applications. The scientific market represents only a very small percentage of the computing market as a whole - the commercial market is much larger and much more lucrative; it seems that if parallel computing wants to expand it needs to move into centre stage of the commercial market and to provide the underlying technology.

Currently as the above brief comments indicate the problem with parallel computing is software, the manufacturers are displaying confidence in the provision of hardware and that help is needed in the software field. There are some encouraging signs in that area for example, HPF and MPI. However, they make take some time to realise their full potential as the solutions for the implementation problems for compilers based on HPF are required.

The urgent problem for parallel computing is therefore software - a lot is known about hardware but users need to be able to write software easily. The medium term problem is the need to expand the market and user bases, the overall problem is the need for a model of parallel computation to help unify and sustain research and development of parallel computing.

References
e-mail: r.perrott@qub.ac.uk

Supercomputing: where have we been? where are we now? where are we going?

A four dimensional parallel integration algorithm is discussed, along with its CRAFT implementation on the T3D. Attention is paid to effective load balancing and, in particular, to reducing communication to a minimum. The CRAFT implementation is found to scale well up to at least 256 nodes.

4-D QUADRATURE IMPLEMENTED IN CRAFT

BY ROBERT M. SINCLAIR, D-MATH, ETHZ, CH-ZURICH

Un algorithme d’intégration du 4ème ordre est présenté, ainsi que son implémentation sur le T3D. L’effort se porte sur la répartition de charge, et en particulier sur la réduction des communications au strict minimum. L’implémentation CRAFT donne de bons résultats jusqu’à un maximum de 256 nœuds.
INTRODUCTION

The program to be discussed here is being used to produce guaranteed upper and lower bounds for integrals appearing in the renormalization group flow of a two dimensional electron gas [1]. These bounds will be used to study the correctness of already existing integration routines (similar to those in [2]), which will themselves be used to actually calculate the flow. This validation is computationally much more intensive than the algorithms it is designed to check, and requires quite different strategies.

When choosing an integration algorithm, the number of dimensions plays an important role. For low dimensional integrals, good general purpose adaptive algorithms are available. For high dimensional integrals, Monte-Carlo is most often the only choice. Four dimensions is at the boundary between these, requiring special considerations. We have decided to concentrate on local adaptive quadrature (where the decision to stop subdividing is made locally, see[3]), since the storage requirements of global algorithms applied to four dimensions are prohibitive. Such local strategies most naturally call for recursive subdivision. For one dimensional integration, one would write:

```
procedure integrate(f,a,b):
    approximate b f(x) dx;
    if (error tolerable) return(approximation);
    else return (integrate(f,a,(a+b)/2) + integrate(f,(a+b)/2,b));
```

The most obvious form of load balancing then consists of passing subdomains of integration from busy to idle processors as they are created during subdivision:

```
procedure rec_integrate(f,a,b):
    approximate b f(x) dx;
    if (error tolerable) return(approximation);
    else if (another processor idle) pass on (f,a,(a+b)/2);
    return(rec_integrate(f,(a+b)/2,b));
    else return (rec_integrate(f,a,(a+b)/2) + rec_integrate(f,(a+b)/2,b));
```

THE STACK-BASED PARALLEL ALGORITHM

Previous work on the Intel Paragon [4] had shown, however, that such algorithms do not scale well. The reason for this is that processors spend most of their time near the ends of branches of the subdivision tree. The newly created subdomains passed on are on average quite close to the end of their branch. An idle processor receiving such a subdomain therefore tends to finish its new work quickly. As a result, the communication overhead becomes unnecessarily high. The problem is particularly serious because the communication network is flooded with short messages (such as «I need work»), meaning that the message latency time rather than the network’s bandwidth is the limiting factor. The very low latency time of the T3D network is of course quite helpful in this respect, but the algorithm is nonetheless in need of improvement.

The solution to the problem is for busy processors to pass on the largest «untouched» subdomains they can to the idle processors. Busy processors can still integrate recursively, but should have access to their own recursion tree, to be able to pass on the largest untouched branches when needed. This requirement forces one to implement recursion explicitly, with a stack. Each processor now has a local stack. It takes subdomains from the top of its stack, and also puts newly created ones there. If a subdomain must be passed on to an idle processor, however, this is taken from the bottom of the stack, where the largest untouched subdomains are to be found. Idle processors are then given, on average, larger sized branches to process, occupying them longer, and reducing the volume of communication.

IMPLEMENTATION IN CRAFT

The above algorithm has been implemented using both local and global stacks in CRAFT [5]. The global stack plays the role of a reservoir of subdomains of integration, through which the passing of subdomains occurs. If an idle processor finds the global stack to be empty, it signals this to the busy processors. One busy processor will then write to the global stack from the bottom of its local stack as described above. The CRAFT code has the following form:

The rather redundant looking double negation in the construction «if (.not.test_lock(no_contribution_needed))» is actually a primitive implementation of a labelled critical section. The function test_lock returns the value of its argument, simultaneously setting it to .true. in an atomic operation. The body of the if statement is only executed if the value of no_contribution_needed was previously .false.. The first processor to do this also sets the value of the lock to .true., effectively locking out other processors.

What is not clear in the above program skeleton, is how the program decides when all the work has been done. In the CRAFT implementation discussed here, this was done by keeping track of the fraction of the original domain of integration (a shared real variable) already treated (ie. subdivision has stopped there). Access to this shared variable was restricted using a lock. The program knows it is finished when the value of this shared variable becomes 1.

The CRAFT implementation scales quite well. The following times are for a fixed problem size, n is the number of processors, T(n) the time in seconds, S(n) the speedup, and E(n) the efficiency. The data for n=1 refers to a version of the program with neither a global stack nor any calls to CRAFT routines. The program was compiled using version 6.2.1.0 of cft77 with the default optimization level.

<table>
<thead>
<tr>
<th>n</th>
<th>T(n)</th>
<th>S(n)</th>
<th>E(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1565</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>523.14</td>
<td>1.00</td>
<td>0.95</td>
</tr>
<tr>
<td>4</td>
<td>3147.54</td>
<td>3.75</td>
<td>0.95</td>
</tr>
<tr>
<td>8</td>
<td>1566.72</td>
<td>7.54</td>
<td>0.94</td>
</tr>
<tr>
<td>16</td>
<td>292.25</td>
<td>14.60</td>
<td>0.83</td>
</tr>
<tr>
<td>32</td>
<td>291.98</td>
<td>29.90</td>
<td>0.82</td>
</tr>
<tr>
<td>64</td>
<td>190.34</td>
<td>59.54</td>
<td>0.93</td>
</tr>
<tr>
<td>128</td>
<td>108.07</td>
<td>115.70</td>
<td>0.90</td>
</tr>
<tr>
<td>256</td>
<td>53.34</td>
<td>221.39</td>
<td>0.87</td>
</tr>
</tbody>
</table>
program integ 
common /global/ global_stack,...
real global_stack(8,...)
CDIR$ SHARED global_stack(:,:block),...
common /local/ local_stack,...
integer notribution_needed
CDIR$ SHARED no_contribution_needed
integer contribution_made
CDIR$ SHARED contribution_made
CDIR$ MASTER
    call set_lock(no_contribution_needed)
    call clear_event(contribution_made)
    initialize global stack (domain of integration)
CDIR$ END MASTER
    initialize local stack (empty)
idle if (all work done ) goto fin
CDIR$ CRITICAL
    if (global_stack_length.eq.0) then
        call clear_lock(no_contribution_needed)
        call wait_event(contribution_made)
    if (all work done ) goto fin
    call clear_event(contribution_made)
        copy an entry from global to local stack
CDIR$ END CRITICAL
busy if (local_stack_length.eq.0) goto idle
approximate integral on top of local stack
if (it's worth subdividing ) then
    subdivide (put new subdomains on top of local stack)
else
    add approximation to local sum
    goto busy
fin call set_event(contribution_made)
CDIR$ MASTER
    add up local sums
    print out results
CDIR$ END MASTER

CONCLUSION

We have presented a successful CRAFT implementation of an unstructured algorithm. Our impression of CRAFT is a quite positive one, particularly since it allows one to write noncritical portions of code in a natural way (no need for explicit message passing), but does not preclude optimization where it is required. This tends to result in readable code, which is of course of great value when debugging or even just updating such a program.

It is also worth pointing out that adaptive integration resembles parallel synchronized branch and bound algorithms in several respects. The interested reader is referred to [6,7] for load balancing strategies applied to that problem.

ACKNOWLEDGEMENTS

This work is part of a PhD at the Swiss Federal Institute of Technology in Zürich (ETHZ), under the supervision of Prof. E. Trubowitz. We are grateful to the Swiss Federal Institute of Technology in Lausanne (EPFL) for granting us time on the T3D there.

BIBLIOGRAPHY


NUMERICAL SIMULATION OF THREE-DIMENSIONAL TIME-DEPENDENT BUOYANT FLOWS ON THE T3D

BY ENRICO NOBILE, LUCA ONESTI, Department of Naval Architecture, Ocean and Environmental Engineering, University of Trieste, I-Trieste & MARCO VOLI, Centro di Calcolo Elettronico dell'Italia Nord-Orientale, I- Casalecchio di Reno

Une procédure numérique est décrite pour simuler à grande échelle des écoulements gravitationnels, tridimensionnels et transitoires. L’algorithme numérique, basé sur une méthode de projection de volumes finis, est capable de traiter des maillages cartésiens arbitrairement étirés dans les 3 directions, avec des conditions aux limites très variées. De plus, il tient compte des obstacles et des surfaces à textures au moyen de la technique de l’immersion de la limite. La parallélisation de tout l’algorithme sur le T3D au moyen du modèle de programmation CRAFT a été rendue possible par une structure soigneusement réfléchie des données. En particulier, l’utilisation des modèles de partage des tâches et de parallélisation implicite des données a donné de bons résultats en ce qui concerne l’accélération et l’extensibilité, même si la performance sur un seul PE est toujours décevante. Nous présentons les valeurs des temps de calcul et du l’accélération pour des grilles allant de 100’000 à 2 millions de cellules.

Nous concluons que les performances prometteuses obtenues sur le T3D vont permettre la simulation numérique directe (DNS) d’écoulements gravitationnels de grand intérêt pour les ingénieurs. Ces simulations ouvrent un potentiel important d’approfondir les connaissances dans de tels écoulements et de développer des modèles de turbulence statistique.
We describe a numerical procedure for large-scale simulation of three-dimensional, time-dependent buoyant flows. The numerical algorithm, based on a time-accurate, finite volume projection method, can handle Cartesian grids arbitrarily stretched in all three directions, and can deal with a large variety of boundary conditions. Furthermore, it can account for obstacles and textured surfaces by means of the immersed boundary technique. Careful design of the data structure has allowed the parallelization of the entire algorithm on the T3D by means of the CRAFT™ programming model. In particular, the utilization of the worksharing and data parallel implicit models, has given good results in terms of speed-up and scalability, though the sustained single-PE performance is still unsatisfactory. Values of the computing time and of the speed-up are presented for grids ranging from about $1 \times 10^5$ cells up to $2 \times 10^6$ cells.

It is concluded that the promising performances obtained on the T3D will enable the Direct Numerical Simulation (DNS) of buoyant flows of engineering relevance. These simulations will provide a more complete knowledge of such flows, and will give reference solutions for improvements and developments of engineering-type, statistical turbulence models.

**INTRODUCTION**

Heat transfer in buoyancy-dominated flows is of fundamental importance in several engineering applications. Relevant examples include passive solar heating, cooling of electronic equipment, crystal growth, chemical vapor deposition (CVD) and phase-change processes.

In these applications the flow is frequently unsteady, transitional and even turbulent, and is characterized by the existence of well organized structures, with length-scales comparable to the size of the domain [1]. These structures often dominate the entire flow field, and strongly affect turbulent transport, mixing and entrainment of fluid. To further complicate matters, in most circumstances of practical relevance, there is the coexistence of regions of turbulent and laminar behaviour.

With relevance to statistical turbulence models, commonly used in industrial Computational Fluid Dynamics (CFD) applications, such features, exhibited by buoyancy-dominated flows, cannot easily be described, at present, by eddy-viscosity models, nor by second-moment closures, as pointed out by Nobile [2] and Hanjalic [3].

Therefore, although some achievements have recently been obtained, there are unresolved questions and controversial issues for single-point turbulence modelling in buoyant flows. Assuming that, for the forthcoming years, second-order turbulence models, either in their complete differential form, or in simplified algebraic expression, will remain the fundamental predictive tools for practical engineering purposes, it can be said that improvements and extensions of existing models will be beneficial to CFD users dealing with problems where buoyancy plays a fundamental role.

Experimental tests at high values of Grashof (Gr) numbers, important for the validation of existing and proposed turbulence models in buoyant flows, are, however, quite expensive to run, and measurements of the fluctuating quantities are still limited in their scope and accuracy [4, 5, 6].

In analogy with constant properties (forced convection) flows, where Direct Numerical Simulation (DNS) of turbulent flows have given access to detailed knowledge of the flow physics and also contributed to better formulation of turbulence models [7], the DNS of buoyant flows is an important tool to acquire an in-depth understanding of the pertinent physics underlaying the phenomena. However, DNS of turbulent buoyant flows has been limited, up to now, to canonical problems like Rayleigh-Bénard convection flow between two horizontal surfaces [8], to two-dimensional simulations [9, 10, 11], or to low values of the Gr number [12].

The main difficulties associated with the simulation of such flows are:

- Presence of wide range of time-scales which requires long-time integration with small-values of the time-increment;
- For vertical thermally-active surfaces, like side-heated enclosures, there is the formation and growth of thin boundary layers along these walls, which in turn call for the adoption of non-uniform grids;
- Existence of outlets, where the flow leaves the domain, which need to be properly modelled, in order to avoid artificial influences on the solution.

This study describes a numerical procedure specifically designed for large-scale simulation of 3D, time-dependent and turbulent buoyant flows, which is aimed to overcome these difficulties. The algorithm is based on a time-accurate, finite volume implementation of a 2nd-order projection scheme. At every time-step, the transport equations for momentum and energy are solved by the Approximate Factorization method, and mass continuity is enforced by the solution of a Poisson equation for the auxiliary variable $\phi$ (pseudo-pressure). The solution of this Poisson equation, which represents the most time-consuming part of the algorithm, is efficiently and accurately obtained by a fast direct Poisson solver, based on Matrix Decomposition [15, 16]. The procedures guarantees strict mass conservation, can be used in conjunction with a large variety of boundary conditions, and can handle Cartesian grids arbitrarily stretched.

In addition, we have verified that the numerical procedure can handle, with very little overhead, multiple obstacles and textured surfaces by means of the immersed boundary technique [17].

The computing power usually required for conducting numerical studies of 3D, time-dependent incompressible flows, makes the use of Massively Parallel Processors (MPP) very attractive [18].

The computer program has therefore been ported on the T3D, as soon as a 64 Processing Elements (PES) configuration was available at CINECA, (Bo), Italy. The parallelization has been obtained by means of the CRAFT™ programming model. In particular, the utilization of the worksharing and data parallel implicit models, has given good results in terms of speed-up and scalability, although the sustained single-PE performance is still unsatisfactory. Values of the computing time and of the speed-up are presented for grids ranging from about $1 \times 10^5$ cells ($5 \times 10^5$ degrees of freedom, d.o.f.) up to about $2 \times 10^6$ cells ($10^7$ d.o.f.). The performance statistics are reported for the Poisson solver, and two selected model problems.

---

1 Sometimes referred to as Fractional-step method, i.e. see [13,14].
i.e. the natural convection flow in a side-heated 3D cavity, and the mixed convection in a bottom-heated horizontal rectangular channel. Compared to a single-processor Cray C90, where the code vectorizes well, the use of 64 PEs T3D reduces the CPU time by a factor in the range (1.7-2.3) for the model problems with O(10⁶) cells. The time is reduced by a factor (5.6-6.6) when using 256 PEs. The promising performances obtained on the T3D, are already enabling the numerical simulation of buoyant flows of practical relevance. These simulations will provide a more complete knowledge of such flows, with particular reference to the transition mechanisms, and will give reference solutions for improvements and developments of engineering-type, statistical turbulence models.

PROBLEM DESCRIPTION AND MATHEMATICAL FORMULATION

The geometry and boundary conditions are illustrated, for the model problems, in figure 1.

![Figure 1 - Geometry of the problems: (a) natural convection in a side-heated enclosure; (b) mixed convection in a horizontal rectangular channel.](image)

The configuration depicted in figure 1-a is the natural convection problem in a cubical enclosure, with two differentially heated, isothermal vertical walls, at temperature $T_h$ and $T_c$ and all other walls perfectly adiabatic.

The geometrical configuration of the thermally developing, mixed convective flow in a horizontal, rectangular channel is shown in figure 1-b. In this case, the two horizontal surfaces are isothermal, with higher temperature, $T_h$, for the bottom wall. The vertical walls are assumed perfectly adiabatic, and there is an inflow and an outflow.

It is assumed that the flow is described by the three-dimensional, unsteady Navier-Stokes equations, embodying the Oberbeck-Boussinesq approximation. The equations are made dimensionless using the following reference quantities: $H$ for length, $U_b = (gH (T_h - T_c))^{1/2}$ for velocity, $\Delta T = (T_h - T_c)$ for temperature, and $\tau_{ref} = H^2/(\nu \Delta T)$ for time, where the Grashof number, $Gr$, the Prandtl number, $Pr$, and the Rayleigh number, $Ra$, are defined as:

$$Gr = \frac{g \beta H^3 \Delta T}{\nu^2}$$
$$Pr = \frac{\nu}{\alpha}$$
$$Ra = Pr \times Gr$$

For the mixed convection problem the Reynolds number, $Re$, is:

$$Re = \frac{U_{ref} H}{\nu}$$

where $U_{ref} = \int_A u \; dA / (H \times W)$ is the mean velocity in the channel.

With the chosen set of reference quantities the non-dimensional governing equations for mass, momentum and energy are:

$$\nabla \cdot \mathbf{u} = 0$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} = -\nabla p + \left(\frac{1}{Gr^2} \right) \nabla^2 \mathbf{u}$$

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \left(\frac{1}{Gr^2 Pr} \right) \nabla^2 T$$

where $u(x,y,z)$ is the velocity, $T$ is the temperature and $p$ is the pressure, divided by the product $\rho U_{ref}^2$.

A detailed description of the initial and boundary conditions, for both model problems, is reported in [19,20]. It is worth mentioning that, in order to better reproduce some experimental tests [21], in the calculations the presence of an isothermal (cold) section upstream the heated part of the channel was taken in account, as shown in figure 1-b.

THE NUMERICAL PROCEDURE

A complete description of the numerical method, reported in [22], is beyond the scope of this article, and therefore only the relevant features are summarized here.

The Navier-Stokes and energy equations, eqns. (1-3) were solved by a finite volume, 2nd order projection scheme, as described by Gresho [14]. The numerical integration from the time instant $t^n$ to $t^{n+1} = t^n + \Delta t$ involves the following steps:

1. Calculation of the new temperature field:

$$\left[ \mathbf{T}_{n+1} - \mathbf{T}^n \right] = \frac{\Delta t}{2} \left( \frac{1}{Gr^2 Pr} \right) \nabla^2 \left[ \mathbf{T}_{n+1} - \mathbf{T}^n \right]$$

$$\Delta t \left[ 3 \mathbf{H}^n - \mathbf{H}^{n+1} \right] + \Delta t \left( \frac{1}{Gr^2 Pr} \right) \nabla^2 \mathbf{T}^n$$

where $\mathbf{H}^n = -\nabla \cdot (\mathbf{u} \mathbf{T})$.

2. Calculation of the intermediate (tentative) velocity field $\mathbf{U}^{n+1}$:

$$\left[ \mathbf{U}_{n+1} - \mathbf{U}^n \right] = \frac{\Delta t}{2} \left( \frac{1}{Pr^2} \right) \nabla^2 \left[ \mathbf{U}_{n+1} - \mathbf{U}^n \right]$$

$$\Delta t \left[ 3 \mathbf{H}^n - \mathbf{H}^{n+1} \right] + \Delta t \left( \frac{1}{Pr} \right) \nabla^2 \mathbf{u}^n$$

$$- \Delta t \nabla p^n + \mathbf{T}^{n+1}$$
3. Calculation of the auxiliary variable $\phi$ (scalar potential or pseudo-pressure) from the Poisson’s equation:

$$\nabla^2 \phi = \nabla \cdot \mathbf{u}^n$$

with $\partial \phi / \partial n = 0$ at all boundaries\(^2\), where $\mathbf{n}$ is a vector normal to the boundary.

4. Update (i.e. project) the tentative velocity $\mathbf{u}^{n+1}$ in order to obtain a divergence-free velocity field $\mathbf{u}^n$ by the relation:

$$\mathbf{u}^n = \mathbf{u}^{n+1} - \mathbf{v} \phi$$

5. Compute the new pressure field at $t^{n+1}$ from:

$$p^{n+1} = p^n + \frac{3}{2} \frac{\phi}{\Delta t}$$

In equations (4) and (5), temporal discretization is done with the Crank-Nicolson scheme on the viscous (diffusion) term and the Adams-Bashforth scheme on the convective term.

The spatial derivatives, which appear after expressing the discretized equations in finite volume form, are computed using central difference relations for non-uniform grids. Accordingly, the procedure is 2nd order accurate in both time and space.

The transport equations (4) and (5) are solved by the Approximate Factorization method. In the case of equation (4), for example, the application of this method can be summarized as follows.

Defining: $\Delta T^{n+1} = T^{n+1} - T^n$;

$$k = \frac{1}{G^{n+1} F}$$

equation (4) can be written, in compact notation, as:

$$\left[ \mathbf{A}^{n+1} + \frac{\Delta t}{\Delta z} \mathbf{K}^{n+1} \Delta T^{n+1} \right] =$$

$$\frac{\Delta t}{\Delta z} \left[ 3 \mathbf{H}^{n+1} + \mathbf{H}^{n+1} \Delta \mathbf{v}^{n+1} \right] + \Delta t \mathbf{K}^{n+1} \mathbf{T}^{n+1}$$

The left-hand side of equation (9) can be approximated in the following way:\(^3\):

$$\left[ 1 - \frac{\Delta t}{\Delta z^2} \frac{\partial^2}{\partial z^2} \right] \left[ 1 - \frac{\Delta t}{\Delta z^2} \frac{\partial^2}{\partial z^2} \right]$$

$$\left[ 1 - \frac{\Delta t}{\Delta z^2} \frac{\partial^2}{\partial z^2} \right] \Delta T^{n+1} = \text{rhs}$$

where rhs is the right-hand side of equation (9). This equation can be split in the following set of simpler, sequential equations:

$$\left\{ \begin{array}{l}
1 - \frac{\Delta t}{\Delta z^2} \frac{\partial^2}{\partial z^2} \Delta T^z = \text{rhs} \\
1 - \frac{\Delta t}{\Delta z^2} \frac{\partial^2}{\partial z^2} \Delta T^x = \Delta T^z \\
1 - \frac{\Delta t}{\Delta z^2} \frac{\partial^2}{\partial z^2} \Delta T^y = \Delta T^x
\end{array} \right.$$

Each equation, in (11), requires the solution of a large set of independent, constant-coefficient, symmetric tridiagonal systems. These subtasks, therefore, can be very efficiently parallelized.

For example, the data and work distribution for the third of (11) – i.e. solution along $z$ – is illustrated in figure 2.

The Poisson-Neumann problem, equation (6), is accurately and efficiently solved by a novel, direct Fast Poisson solver based on matrix decomposition. The choice of a direct solver has been motivated by the following considerations:

- The solution of the equation has to be obtained, on large 3D grids, at every time-step, and thus this requirement leaves out standard iterative solvers (i.e. SOR, ADI);
- Since for incompressible, unsteady flow simulations mass-conservation has to be tightly satisfied, direct compact solvers are particularly attractive;
- The parallelization of a direct solver is easier, for MPP computers, than other fast Poisson solvers (i.e. multigrid).

The efficiency of the (pseudo-)pressure solution is of paramount importance in unsteady, 3D, incompressible flow simulations. The pressure solution demanded about (50-60)% of the CPU time in our calculations. With a less efficient Poisson’s solver, this part of the computation can overwhelm the remaining semi-implicit calculation of the transported variables, and thus make the simulations unfeasible. The complete derivation of the matrix decomposition algorithm of the 3D Poisson solver for non-uniform, finite volume grids is reported in [22] and is not repeated here. It is worth mentioning that, in comparison with a similar algorithm described by Babu and Korpela [16] for finite difference discretization, the one we have developed has the following advantages:

1. It does not require analytical transformations in order to deal with non-uniform grids;
2. Since the coefficients are obtained by finite volume discretization, the integral constraint\(^4\), i.e. of the same order introduced by the temporal discretization, is identically satisfied. This avoids the practice, described in [16], of re-adjusting the rhs at every time-step;
3. The Euclidean norm of the residual, even for highly stretched grids, is constantly of the order of machine round-off (say 10\(^{-20}\)), a result not possible for finite difference discretizations, as

---

\(^2\) More exactly, at all boundaries where the $\mathbf{v}$ component of the velocity is specified.

\(^3\) The error introduced is of the order $(\Delta t)^3$, i.e. of the same order introduced by the temporal discretization.
reported in [16].

The end result is that, since mass conservation is constantly satisfied to machine accuracy, long-term integration of 3D unsteady flows at high Gr numbers are accurate and reliable.

In all the simulations the time-step $\Delta t$ is allowed to dynamically adapt during the calculation, and chosen to satisfy everywhere on the grid the following stability requirement:

$$\Delta t \leq \frac{\max_{x,y,z} \left| \frac{\partial H}{\partial x} + \frac{\partial M}{\partial y} + \frac{\partial N}{\partial z} \right|}{\max_{x,y,z} \left( \alpha T \right)}$$

(13)

**CODE VALIDATION**

**STEADY NATURAL CONVECTION IN A SIDE-HEATED 3D CAVITY**

The accuracy of the numerical model has been verified for a value of the Grashof number, $Gr$, of $1 \times 10^6$, and a value of the Prandtl number, $Pr$, of 0.71. For these values of the governing parameters the flow is steady, and numerical calculations have been performed, among others, by by Fusegi et al. [23] and Janssen et al. [24]. In particular, the results obtained by Janssen et al. are probably, to the author’s best knowledge, the most accurate, since they have been obtained on grids, up to $120^3$.

It is worth to remember that our results have been obtained as the true time-asymptotic solution of the flow, following a long transient from isothermal, quiescent conditions.

Table 1 reports the comparison, between the present results and those of other authors, of some characteristic quantities, namely:

- The space-averaged Nusselt number, $Nu$, on the hot vertical wall:

$$Nu = -\frac{1}{H} \int_0^H \left( \frac{\partial \bar{T}}{\partial x} \right)_{x=\frac{H}{2}} dz$$

(14)

- The maximum vertical velocity $w_{\text{max}}$ at $(x,H/2,H/2)$.

- The maximum horizontal velocity $u_{\text{max}}$ at $(H/2,H/2,z)$.

- The thermal stratification at the centre of the cavity:

$$S = \frac{H}{\Delta T} \frac{\partial T}{\partial x}$$

(15)

From table 1 it can be seen that the present algorithm is very accurate, with differences in $Nu$ value, with the results of Janssen et al., of less than 0.2%. The agreement for the other quantities is also very favourable.

**TIME-DEPENDENT POISEUILLE-BENARD FLOW IN A 2D CHANNEL**

This problem has been extensively studied by Evans and Paolucci [25]. At $Re=10$, $Pr=2/3$ and $Gr=1.5 \times 10^4$, the flow is periodic in time, and consists of traveling transverse waves superimposed to the longitudinal mean flow. Therefore, this problem is a valid test-case for time-dependent, buoyancy-affected flow problems. The geometry of the problem, together with the boundary conditions, is sketched in figure 3.

The calculations have been performed at two grid resolutions, and the characteristic results are compared with the data in [25] in table 2.

Our results show a good agreement with the benchmark solution [25], despite the fact that Evans and Paolucci used a longer computational domain, and Richardson extrapolation in both space and time. It can be concluded that the algorithm adopted is accurate also for time-dependent flow problems.

**PERFORMANCE STATISTICS**

In the table 2, the performance results are illustrated first for the Poisson solver, and then for the two model problems.
**Numerical Simulation of Three-Dimensional Time-Dependent Buoyant Flows on the T3D**

<table>
<thead>
<tr>
<th>Present results</th>
<th>Evans &amp; Paolucci [25]</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid 30x200</td>
<td>grid 60x400</td>
</tr>
<tr>
<td>t</td>
<td>1.2764</td>
</tr>
<tr>
<td>l</td>
<td>1.4100</td>
</tr>
<tr>
<td>Ut</td>
<td>1.1100</td>
</tr>
<tr>
<td>Nu</td>
<td>2.5049</td>
</tr>
<tr>
<td>(u_{\text{max}})</td>
<td>4.3440</td>
</tr>
<tr>
<td>(y_{l}u_{\text{max}})</td>
<td>0.8167</td>
</tr>
<tr>
<td>(u_{\text{min}})</td>
<td>-2.3878</td>
</tr>
<tr>
<td>(y_{l}u_{\text{min}})</td>
<td>0.1500</td>
</tr>
<tr>
<td>(v_{\text{max}})</td>
<td>5.0452</td>
</tr>
<tr>
<td>(y_{l}v_{\text{max}})</td>
<td>0.5000</td>
</tr>
<tr>
<td>(v_{\text{min}})</td>
<td>-5.0406</td>
</tr>
<tr>
<td>(y_{l}v_{\text{min}})</td>
<td>0.5000</td>
</tr>
</tbody>
</table>

Table 2 — Comparison of results for the Poiseuille Benard problem in a 2D channel; \(Re=10\), \(Pr=2/3\), \(Gr=1.5\times10^4\)

**Direct Poisson Solver**

The performance of the Poisson solver, for two grids, is summarized in Table 3. For comparison purposes, the computing time, on the coarser grid, is given also for a single-processor Cray C90. The CPU time reported does not include the factorization and preprocessing phase, which is done only at the beginning of the calculation. Furthermore, the time spent in this phase becomes negligible for large grids.

It can be noticed, from Table 3, that the speedup obtained stays very close to the theoretical value up to 128 PEs. At that point, the speedup value diverges, and little gain is achieved by further increase of the number of PEs. This behaviour can be explained by observing that, in the Poisson solver, the data are distributed, among PEs, in form of 2D matrices – slabs of cells along the z-axis. Since the number of z-cells is exactly 126 (+2), this explains the decay in efficiency for PE numbers above 128.

**Natural Convection Problem**

Values of the computing time and of the speedup, for a single time-step and two grid resolutions, are reported in Table 4.

<table>
<thead>
<tr>
<th>No. PEs</th>
<th>Theoretical time (s)</th>
<th>Real time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.000</td>
<td>12.325</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.000</td>
<td>6.263</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.000</td>
<td>3.266</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8.000</td>
<td>1.646</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>16.000</td>
<td>0.837</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>32.000</td>
<td>0.420</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>64.000</td>
<td>0.211</td>
<td></td>
</tr>
<tr>
<td>Cray C90</td>
<td></td>
<td>0.767</td>
<td></td>
</tr>
</tbody>
</table>

Table 3 — Speedup and computing time for the Poisson solver.

In comparison with the trend observed in Table 3, it can be noticed that, in this case, the reduction in efficiency for 256 PEs is not as severe as that experienced for the Poisson solver alone. This can be explained by remembering that, as depicted in Figure 2, in the Approximate Factorization solver used for the transported variables, the data are distributed by 1D arrays – lines – and therefore better suited for higher PE numbers.

<table>
<thead>
<tr>
<th>No. PEs</th>
<th>Theoretical time (s)</th>
<th>Real time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.000</td>
<td>24.760</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.000</td>
<td>12.680</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.000</td>
<td>6.470</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8.000</td>
<td>3.460</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>16.000</td>
<td>1.819</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>32.000</td>
<td>0.929</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>64.000</td>
<td>0.491</td>
<td></td>
</tr>
<tr>
<td>Cray C90</td>
<td></td>
<td>1.101</td>
<td></td>
</tr>
</tbody>
</table>

Table 4 — Speedup and computing time for the natural convection problem.
Numerical Simulation of Three-Dimensional Time-Dependent Buoyant Flows on the T3D

An interesting performance indicator, commonly used in unsteady flow calculations, is the CPU time/(time-step x node), or, alternatively, the CPU time/(time-step x degree of freedom). In our case, on the 126³ grid and 256 PEs, these values are 1.31 \(\mu\)s and 0.262 \(\mu\)s, respectively.

**Mixed Convection Problem**

The performance data are summarized in table 5, for two different grid sizes.

For the 254x62x62 grid the CPU time/(time-step x node) is about 1.17 \(\mu\)s for 256 PEs, a value about 10% better than that obtained for the natural convection problem. However, the two values cannot be easily compared, given the different number of cells, and their different distribution along the three directions.

**Example of Flow Physics**

In this section, for illustrative purposes, we present a sample of the results obtained from the simulation of mixed convection in a bottom-heated horizontal channel.

The predictive capabilities of the numerical model are assessed by comparison of the computed results with the experimental measurements of Chiu and Rosenberger [21], obtained for a channel with dimensions of 875 mm (L) x 152.4 mm (W) x 15.8 mm (H). In their experiment, nitrogen was used as the working fluid, and one set of experimental conditions corresponds to Re=42, Pr=0.71 and Gr=6884. For these values of Re and Gr they obtained a steady flow, which consists of a combination of longitudinal rolls and the mean flow. The numerical simulation of this experiment was done using a grid of 126x62x30 cells, with appropriate stretching along the z-axis. In agreement with the experiment, after the initial transient, the steady-state condition was numerically realized after about 6000 time-steps. A comparison of the computed and measured longitudinal velocity profile is given in figure 4, for z=0.5H, and x=12.7H.

![Figure 4 — Comparison of the predicted streamwise velocity profile with the experimental data of Chiu and Rosenberger [21](a)](image)

![Figure 5 — Visualization of the temperature isosurface T = 0.5 for the mixed convection in a large aspect-ratio channel: (a) - Re = 20, Gr = 6884; (b) - Re = 10, Gr = 6884; (c) - Re = 10, Gr = 10000.](image)

The profile correspond to a section within the thermal development length. The close agreement between measurements and calculations suggests that the salient features of the problem have been properly modelled. At higher values of Gr and/or lower values of Re, the shear, induced by the imposed mean flow, is not anymore sufficient to stabilize the flow, which bifurcates to an unsteady solution. Figure 5 illustrates the different pattern of the temperature field, represented by the dimensionless T=0.5 temperature isosurface, at different values of Re and Gr.

Figure 5-a shows the presence of steady, longitudinal rolls, while figure 5-b, relative to a time-instant, indicates the simultaneous presence of both longitudinal rolls, in reduced number, and transverse, traveling (unsteady) waves. Finally, figure 5-c, which represents an instantaneous solution, depicts the chaotic behaviour of the flow, characterized by the unsteady modulation – snaking – of the longitudinal rolls. A further reduction in the number of rolls is clearly visible.

**Conclusions**

In this work a numerical algorithm for large-scale simulation of time-dependent, 3D buoyant flows is presented. The algorithm, based on a conservative finite volume, 2nd order projection scheme, has been proved accurate and reliable for illustrative examples of three-dimensional, natural and mixed convection flows.

The code has been ported to the T3D by means of the CRAFT© programming model, that has given us the opportunity to exploit, with minimal investment, the T3D hardware.
Good performance data, in terms of speed-up and scalability, have been obtained for the problems considered, using up to 256 PEs. While the results obtained are still unsatisfactory in terms of single-PE performance, preliminary optimization, within CRAFT, is in progress, and is paying off tremendously. As an example, the use of the single-PE, optimized SGEMM or SGEMMV routines, instead of the MATMUL intrinsic, for matrix-vector multiplication in the Poisson solver, has reduced the CPU time (126\textsuperscript{3} grid and 64 PEs) from 3.62 s to 2.11 s (SGEMM) and 1.088 s (SGEMMV).

It is concluded that the numerical model described, implemented on the T3D, represents a useful tool for the study of large-scale, time-dependent and possibly turbulent, buoyancy-dominated flows. Its flexibility, in terms of boundary conditions and internal obstacles, will be further exploited for the study and optimization of textured, enhanced heat-transfer surfaces.

**Acknowledgements**

This work was partially funded by *Ministero dell’Università e della Ricerca Scientifica e Tecnologica* (MURST 40% and 60%). The first author gratefully acknowledges the T3D computing time provided by CINECA (BO). Thanks are due to R. Ansaloni for assistance with the benchmarking.

**References**


A cause de la quantité importante de données qui peut être générée par des applications sur des systèmes massivement parallèles comme le T3D, il est souvent impossible ou peu pratique de sauver les résultats du calcul sur disque pour être visualisés ultérieurement. Pour résoudre ce problème, un outil de visualisation en temps réel, TPview, a été développé. Cet outil envoie des données distribuées entre les processeurs du T3D via des sockets UNIX à une station graphique afin de les traiter pendant la simulation. L’outil utilise la librairie de communication interprocessus pour la manipulation et l’affichage des données par TECPLLOT. Une application dans le domaine de la dynamique des fluides servira d’exemple d’utilisation de cet outil, pour lequel diverses considérations liées à la performance seront abordées.

Due to the massive amount of data that can be generated by application codes on high-performance parallel systems such as the T3D, it is often not possible or practical to store the computational results on disk for post-processing. To overcome this problem, an on-line visualization tool, TPview, has been developed. Data distributed across multiple processors of the T3D are sent via UNIX sockets to a remote graphics workstation for post-processing during the computation cycle. The tool employs the inter-process communication library for processing of the data by the TECPLLOT visualization software. An example of the use of this tool for a Computational Fluid Dynamics application is presented and various issues associated with its performance are addressed.

**INTRODUCTION**

The productivity of high-performance computing for the numerical simulation of scientific applications can be greatly increased through the availability of easy-to-use visualization tools. In fields such as Computational Fluid Dynamics (CFD), access to graphical representations of numerical results is imperative in order to verify solution correctness and to provide physical insights into the computed flows. This is particularly critical for numerical simulations performed on high-performance parallel computer systems such as the T3D, due to the massive amount of data that is generated during the solution procedure.

The process of visualization can generally be divided into five discrete phases:

1. computation,
2. data selection or filtering,
3. data transfer (if necessary),
4. rendering,
5. display.

Different visualization approaches can be categorized according to how each of the above phases are accomplished. In particular, for a heterogeneous computer network such as considered in the present study, it is important to distinguish on which platform each phase is to be performed. This study is concerned with the visualization of datasets that result from large-scale scientific simulations, with the computation phase performed on a high-performance parallel computer system. These datasets are thus generally distributed across multiple processors.

Two different options exist for the data selection, which may be undertaken either on the parallel system or on a remote system. Since the data to be visualized is distributed across the processors, this phase can generally be accomplished in parallel involving only a small computational overhead. In addition, if the rendering phase is to be performed on a remote system, any data selection or filtering is best undertaken on the computational system in order to reduce the quantity of data to be transferred and hence the communication overhead.

At the other end of the visualization process, the computational results could be displayed either on a framebuffer connected directly to the parallel system, or on an X-Windows display. The use of a framebuffer provides potentially higher visualization throughput. However, while a framebuffer would generally be located at a distance from the user’s office, the X-Windows display can be placed on the user’s desk. In addition, a framebuffer may not be available on the parallel system. Indeed, the T3D system at the EPFL employed for the present study does not have an attached framebuffer. It is therefore preferable that the results be visualized – as is the case for the present study – on an X-Windows display.

Three possibilities exist for the rendering phase: either on the parallel system, on the front-end of the parallel system (if one exists), or on a remote computer system. Two problems are encountered if the parallel system is to perform the rendering phase. First, this will increase the total execution time of the simulation, and this overhead could be quite large depending on the complexity of the image to be rendered. The second problem is the potential lack of suitable volume rendering tools. (For example, despite current software developments [1], there are presently no suitable volume rendering tools for the T3D.) For the present study, rendering is performed on a remote graphics workstation, rather than on the front-end of the parallel system, for the following reasons:

- there are more visualization packages available on workstations,
- and there is a better chance that a suitable one is already installed,
- whereas the front-end of a parallel system is often heavily loaded (as is the case for the CRAY Y-MP system at the EPFL), a dedicated graphics workstation is generally available.

Since the data selection and rendering phases are performed on different computer systems, a data transfer phase must also be undertaken. Various possible options will be discussed in more detail in the following section.
ON-LINE VISUALIZATION

The most common approach to scientific visualization currently employed is termed batch visualization. This consists of saving a dataset, or many datasets at given times within the simulation, into disk files. After the simulation has completed, the data selection is performed as a post-processing phase, either on the computational or visualization system. Following the data transfer phase — generally via ftp file transfer — the datasets are imported one at a time into a visualization package. The batch visualization approach has many drawbacks, including:

- the time and effort needed to manipulate and transfer the data,
- the amount of disk space required to store all the datasets,
- the unnecessary waste of resources if, after a simulation of many minutes or hours, the results are found to be incorrect,
- the time required to transfer all the datasets from the computational system to the visualization system.

The static nature of this approach which does not facilitate the animation that is often required.

An alternative approach — adopted in the present study — is on-line visualization, for which computation and visualization are performed concurrently. Here one dataset is processed for visualization while the next dataset is being computed. This paper describes an on-line visualization tool, TPview, that has been developed to allow data computed on the T3D to be visualized on a remote workstation. The goals of this development were to build a versatile tool which would have as little impact as possible on the numerical simulation. This implies that the execution time overhead due to the visualization should be minimal, and that the user should be required to make as few changes as necessary to his code. Before presenting an overview of the TPview tool, specific details for the data selection, data transfer and rendering phases are outlined.

DATA SELECTION

It is often only necessary, particularly for large datasets, to view a subset of the data from the numerical simulation. A versatile method is therefore needed to indicate which subsets of the data are required to be sent. This is performed using a concept borrowed from the PORTAL communication library developed at the Argonne National Laboratory [2], which has been integrated into a library named TPV. Data selection is undertaken via calls to the TPV library that is linked to the simulation code. This library has the advantage of being rather versatile and unintrusive, with calls to the TPV library being easily added to or removed from the application code.

DATA TRANSFER

For transferring data from the parallel system (the T3D) to the graphics workstation (a Silicon-Graphic Indigo2 Extreme) connected via a Local Area Network (Ethernet), either message passing or UNIX sockets can be used. A description of the use of sockets can be found in [3]. For the TPview tool, sockets have been chosen for the following reasons:

- sockets can be manipulated without any explicit use of the Y-MP front-end; while the system calls are still handled by the Y-MP, they are completely transparent to the T3D,
- message passing libraries need to be installed on each machine involved in the communication, whereas sockets are contained in standard UNIX and are thus available on any UNIX machine,
- while message passing libraries such as PVM or MPI use sockets as its underlying transfer mechanism, additional overhead results from the use of buffers and daemons.

Sending data by sockets is made transparent to the user via calls to the TPV library. The use of sockets distinguishes TPview with other on-line visualization tools, such as PV3 [4], that employ message passing for data communication across the network.

Figure 1 shows the type of transfer rates that are achieved using sockets between the T3D and a workstation on the Ethernet network. It should be noted that these measurements pertain to a socket library written by one of the authors that is not guaranteed to be an optimal implementation. The time taken for various buffer sizes to be sent from the T3D to the workstation and back has been measured. Minimum, maximum and typical timings are shown in Fig. 1. Two points that should be noted from this graph are that transfer rates of up to 4 Mbit/s have been achieved (approximately half of the Ethernet peak bandwidth), and that occasionally delays of up to one second are obtained. While the cause of these delays is unclear, it is suspected that they are due to the load of the Y-MP, as the same type of results can be observed when performing disk I/O from the T3D on the Y-MP.

![Figure 1] Ping-pong transfer times using sockets between the T3D and workstation connected via Ethernet.

RENDERING

A number of scientific visualization packages exist for graphics workstations. For the present development, TECPLOT [5] was chosen for the following reasons:

- it is a widely-used visualization package, especially for CFD applications, and is available on a large number of different computer systems,
- it is the standard visualization package at the fluid mechanics laboratory of the EPFL,
- it is capable of generating the types of plots required, using a convenient input format. (In particular, a dataset can be divided into multiple blocks, which corresponds directly to data distributions on parallel machines where each processor memory contains a block of the whole dataset. This has simplified the implementation of the TPview tool, as a one-to-one relationship can be made between a processor and a block.)
- it is delivered with an inter-process communication (IPC) library which allows a user-written program to use TECPLOT as a...
slave, by sending it commands and by passing run-time data, has been measured to have a higher throughput than commonly-employed Modular Visualization Environments (MVE), such as AVS or Explorer [6]. The use of PORTAL on the T3D [7] also necessitated the development of additional AVS modules for the treatment of multiple blocks.

TPview

A schematic diagram of the layout of the TPview tool is presented in Fig. 2. As introduced above, three different computer systems are employed: a high-performance parallel system (T3D), a graphics workstation for rendering, and a X-Windows display.

![Figure 2 – A schematic diagram of the TPview layout.](image)

The visualization process begins by one processor of the T3D sending a request to the TPview daemon on the remote workstation, which uses the parameters of this request to start the TPview module. This program then starts TECPLOT, initializes a shared memory segment between TECPLOT and itself using the IPC library, and starts a menu module. It then initializes a socket from itself to a T3D processor, and waits for socket connections to arrive from the various sending processors. The T3D application can then continue by performing data selection and by initializing output sockets to the TPview module using the TPV library. At each send iteration, the T3D application makes a library call to send a new data subset via the sockets to the TPview module, which writes this data into the shared memory segment. It then instructs TECPLOT to render the newly received data. At any time during this procedure, the user can interact with the menu module to send any user input to the TPview module, which will then forward it to the T3D using its output socket. All output from the menu module and from TECPLOT can be visualized on any X-Windows display by setting the appropriate value for the DISPLAY environment variable.

The TPV library also has the following features:

- It may be beneficial to regroup the data onto a smaller number of sending processors that are directly involved in transmitting the data.
- 8-byte entities from the T3D are converted into 4-byte entities before sending to the remote machine, in order to decrease the quantity of data to be transferred; since workstations generally use 32-bit IEEE data format, this conversion would be required even if not performed by TPV.

**TPview Example**

To illustrate the use of TPview, a code segment containing calls to the TPV library is shown in Fig. 3. In this example, each processor has a subset (block) of a distributed array containing a multi-component physical variable. The data is stored in a local array called viz_vector containing five two-dimensional variables each of size xdim by ydim. At the initialization step (n_iter = 1), a call is made to TPStart, with the following arguments: an array of PVM task IDs (tids) for the processors participating in the data transfer to the remote workstation, the number of elements in the tids array, and the path of a configuration file required to start TPview. This configuration file contains the name of the machine on which to start TPview and the path of a configuration file on the remote workstation, which TPV needs to configure itself.

```fortran
INCLUDE 'tpview.inc'
INTEGER handle
REAL viz_vector(5,xdim,ydim)
IF (n_iter.EQ.1) THEN
  CALL TPStart(tid,n_tasks,:'/t2/sawley/TPCONFIG')
  CALL TPCreateHandle(handle,TP_REAL,xdim,ydim,1, &
  FIRST_DIM,3,5,viz_vector)
  CALL TDaDataInit(handle,1)
ENDIF
CALL TPSetSendInterval(50)
IF (n_iter.EQ.1) THEN
  CALL TPGetData(handle,n_iter)
ENDIF
CALL TPEnd()
IF (n_iter.EQ.250) THEN
  CALL TPDataSend(handle,n_iter, &
  BITDUMP,"/u/sawley/iter250")
ENDIF
IF (n_iter.EQ.n_iter_max) THEN
  CALL TPEnd()
ENDIF
```

![Figure 3 – Example of a code segment that uses the TPV library.](image)

The second call is made to TPCreateHandle, which involves the data selection phase. The first argument is a handle (i.e., a pointer) to a TPV internal structure defining the selection. The subsequent arguments are parameters of the selection: the data type to be sent (i.e., real, integer, etc.), the X, Y, and Z dimensions of the source array, a flag to indicate the dimension in which the variables can be found in the source array, the number of variables to send from the source array (in Fig. 3, variables 1, 2 and 3 are sent, corresponding to pressure, and the X and Y components of the velocity vector), the size of the first dimension of the source array, and finally a pointer to the source array.

TPGetDataInit takes the information in the handle and sends it in the form of a packet to the remote workstation. At this stage any socket connections are also initialized.

The TPV library routines TPSetSendInterval and TPGetSendInterval define and retrieve the frequency at which data will be sent. This parameter can be adjusted by the user according to his needs.
50 iterations. The default send interval is each iteration.

Following this initialization stage, the application program runs as usual, updating the contents of viz_vector. When it is required that data be sent, a call to TPSendData is made, with an argument specifying the corresponding iteration number. At any time during the numerical simulation, the user may send commands directly to TECPLOT via a call to TPSendCommand. For example, in Fig. 3 a bitdump is requested to be taken at iteration number 250 and written to the file specified as the second argument, on the remote machine. The list of possible TECPLOT commands is contained in Chapter 27 of [5].

When TPview is no longer required, each processor calls TPEnd.

To illustrate the type of graphical interface the user would observe, the code in Fig. 3 has been implemented into a parallel multi-block flow solver [8] to compute the incompressible flow through a turbine cascade. A pressure contour plot with superimposed flow streamlines for a two-dimensional slice through an inter-blade channel is rendered at each step, according to a pre-determined TECPLOT style file. For this example, the computational mesh does not change during the simulation. To avoid unnecessary data transfer, the mesh coordinates are therefore stored in a file on the workstation that is read during TECPLOT initialization. Figure 4 presents the visualization display obtained for time step 450.

In addition to the standard TECPLOT display window, a separate window is observed containing a menu of commands. The iteration number of the currently displayed image is shown on the first line. Next, a slider is available for the user to change the data send frequency as the simulation progresses. The current send frequency iteration number of the currently displayed image is shown on the first line. After each subsequent press the next iteration dataset will be sent and rendered.

Step: pause at the current image (this also pauses the simulation on the T3D) when pressed for the first time. For each subsequent press the T3D continues to deliver the next iteration dataset until stopped again. The current send frequency does not change during the simulation. To avoid unnecessary data transfer, the mesh coordinates are stored in a file on the workstation that is read during TECPLOT initialization.

Pause/Continue: Pause is the equivalent to the first Step press. After pressing Step or Pause, this button becomes Continue. Pressing Continue returns the simulation to normal processing mode (i.e., send data without waiting for a message from TPview).

Quit: close all connections with the T3D, release control of TECPLOT, and exit TPview. After pressing Quit, the simulation continues execution but will no longer send information to TPview. TECPLOT remains open, so the user can manipulate the data from the last image rendered.

**TPview Performance Study**

To determine whether TPview is capable of fulfilling the design goals, a series of timing experiments have been undertaken using the same flow simulation that was performed to obtain the display shown in Fig. 4. The experiments involve varying the number of sending processors directly involved in the data transfer to the remote workstation, measuring the performance of the TPV library, and measuring the influence of TPview on the total execution time of the application code. The simulations were run on 16 processors of the T3D, with each processor containing visualization data subsets of equal size. (Similar results have been obtained for simulations using 4 processors.) The entire dataset represents approximately 160 KBytes. Since 8-byte data is converted into 4-byte before sending, this represents 80 KBytes to be sent at each iteration.

Figure 5 shows the execution times measured for different numbers of sending processors. The times for both the calculation phase only and the total time are shown for the entire simulation (1000 iterations). For the calculation stage, it is seen to be best for all 16 processors to send their own data, followed by 4 to 8 processors inclusive. The gap between the calculation time and total time is the same (~ 6 sec) for all cases, except for 16 sending processors for which the increase is approximately 18 sec. This difference corresponds to the time required to start TPview, which in turn starts TECPLOT and the menu module, sets up the shared memory, and initializes the socket connections. The larger difference observed for 16 sending processors is due to increased socket initialization and termination overheads. It appears that above a certain number of sockets, the Y-MP causes considerable delays. If the simulation should last for many minutes, this overhead may become unimportant. It is suspected that for an even larger number of sending processors, this delay will become a dominating factor, hence the interest in regrouping data onto a subset of processors.

The time taken to send data each iteration is detailed in Fig. 6. For each sending processor, the time taken for each call to TPSendData is measured, as well as the time taken for the write to the socket. Figure 6 shows the maximum of these two values together with the sum of the write times for all processors (total send time). Of note here is that the delays are the smallest when 4 processors send the dataset. While this result seems contrary to the data of Figure 5, the difference can be explained by the fact that when sending with 4 processors, the other 12 are idle, resulting in a load balance problem. This imbalance may in fact be exploited by assigning a larger proportion of the computation to those processors not sending data.
A possible optimization parameter for the T3D is the environment variable MPP_AGENT_SYSCALL_THREADS. This variable, used by mppexec, defines the number of threads available on the Y-MP to handle system calls. When a number of processors are writing to sockets, each perform system calls. If the above-mentioned environment variable is set to 2 (the default for the EPFL T3D system) then writing to at most two sockets can be performed at the same time.

When dealing with large datasets, the time required by TECPLOT to render the image can become overwhelming, and it may therefore be preferable to consider the following options:

- reduce the volume of data to render by only displaying the particular area of interest,
- reduce the complexity of the image to be rendered,
- increase the rendering speed by running TECPLOT on a more performant computer system,
- perform the rendering on the parallel system (subject to software availability),
- perform batch visualization.

The third option may lead the user to consider running TPview and TECPLOT on the Y-MP (if it is not heavily loaded). This would also provide the possibility for significantly faster data transfer, for example using the T3D/Y-MP high-speed channel and the /proc file system to write directly into the memory of a Y-MP process.

**Future Improvements**

In this continuing development work, it is desirable that more functionality be added to the TPview tool to overcome some of its limitations. In particular, the following improvements are to be considered:

- it is necessary to determine the cause of the occasional large send times when using the socket library, and make modifications so that these times are more uniform,
- more menu control is required, e.g., to be able to manipulate the current dataset after having paused at a given image,
- as much as possible of the tuning mentioned in the previous section should be performed automatically,
- the TPV library should be ported to a number of different computer systems,
- the communication protocol employed by the TPV library (e.g., PVM, MPI, SHMEM, sockets for a networked computer system) should be chosen for optimal performance.

**Conclusions**

It is considered that the on-line visualization tool, TPview, that has been developed fulfills the design goals. It has minimal impact on the application code, in terms of both code modification and execution time, while being sufficiently flexible and easy to use. In particular, current TECPLOT users need acquire only minimal additional expertise to employ TPview.

While TPview is not designed to be the optimal tool for all visualization needs, experience to date indicates that it should be very useful for a wide range of scientific visualization. In particular,
it has be found to reduce considerably the inherent difficulty in the analysis of large datasets generated by numerical simulations on a high-performance parallel computer system.

ACKNOWLEDGEMENTS

This study was undertaken within the framework of the Cray Research – EPFL Parallel Application Technology Program, and was performed using the T3D system at the EPFL.

REFERENCES

and training periods were organized by the PATP support group to prepare people for massively parallel programming techniques.

We received the T3D machine, with 128 PEs of 2 Mwords each, by the end of May 1994. It started production on June 2nd, and immediately the PATP group was able to run application programs.

First results of this work were presented at the fall 1994 CUG, in Tours (France), by a number of PATP members. A few months later, all the hardware boards where changed so that the T3D, in a single cabinet model, was configured with 256 PEs with 8 Megawords each. Since then the distributed memory size (16 Gigabytes) is four times that of the Y-MP (4 Gigabytes).

THE ACADEMIC ENVIRONMENT

The manner in which the Y-MP and the T3D were installed had the following major consequences: a fairly large number of users with different needs, skills and goals, wanted to work independently on the Y-MP only, or on both machines.

The Y-MP acts as a CPU server for its own group of users, and also as a front-end for the T3D users. Users are from the EPFL research centres; however, they may also be from external groups, for example other universities or private companies.

The T3D plays a special role: its primary goal is to provide resources to the PATP projects with highest priority. A development and production environment must be provided to all PATP members. Several EPFL computer science departments, institutes and external universities, have independently expressed their interest for T3D projects. A numbers of various users are thus working with the T3D, making connections from Spain, France, Italy, Germany, The Netherlands and Great Britain.

The Y-MP set of users contains about 600 accounts, of which about 100 are active, whereas the T3D has about 130, of which, including the PATP accounts, 30 are accessed daily.

This is to show that the Y-MP and the T3D are heavily loaded, and managing both machines requires specific attention so that services assigned to each can be fulfilled.

ADMINISTRATIVE AND SOFTWARE CONFIGURATION

The general idea was to create two distinct groups of users: the first contains those people allowed to run programs on the Y-MP server only, the second group contains those users allowed to work on the T3D and the Y-MP.

The hardware configuration was organized to allow this scheme to be easily implemented. Disks, channels and other peripherals are linked to the Y-MP in such a way that most of the pure T3D I/O streams are independent from those of the Y-MP. Each machine has its own set of home directories and scratch areas. This method clarifies administrative tasks, improves throughput and best separates the T3D specific problems from those of the Y-MP.

The NQS system contains 2 sets of queues. The first set allows normal Y-MP work, and the second permits jobs to be launched on T3D. All the queues are «pipe-only», so job assignment to a given queue is determined automatically by the requested resource type. Currently, there is only one common batch queue for the 2 sets. Each set, in NQS terms, is defined as a «complex». Thus, a NQS job will be assigned to the T3D complex as soon as the user has specified a T3D resource. For those users allowed to work with the T3D, the access is therefore transparent.

STANDARD USER PROFILES

Interactive profiles and machine access are controlled by the udb user data base. Batch profile limits are also defined in the udb, but the access control, which is only needed for public and non public queues, is done by setting the NQS limits and permissions for each queue.

Current standard profiles on the T3D are the following:

- batch mode: up to 256 PEs (the full T3D capacity), with various time limits;
- interactive mode: 16 PEs and 15 minutes. A few selected users are allowed to use 64 PEs.

We had to enlarge the Y-MP limits for the T3D interactive users, as T3D cross compilers and linkers require about twice the resources the same utilities require for the Y-MP.

Training accounts may use up to 8 processors either in interactive or batch modes.

All the PATP group members are located in a separate EPFL building and have accounts on a local Sparc server of their own. In order to simplify the PATP work, all the users home directories are cross-mounted using NFS between the Y-MP and the local server, and a uniform distributed environment between the 2 machines is provided. So common tasks such as editing or viewing results on the Y-MP may be done remotely.

Y-MP/T3D BATCH PRODUCTION CHARACTERISTICS

With the emergence of large simulations, we observed that many users are chaining their tasks. In other words, it appears that a single NQS batch run will not solve the researchers problems: a large number of similar runs are necessary for complete results. As a consequence, many users insert a qsub command at the end of their job, and resubmit the same job in the same queue. Some sites do not allow this practice, but we think it is useful and convenient for many users.

Thus several independent jobs streams may run at the same time on the T3D, asking for the same amount and type of resources (mostly 32 or 64 PEs). This kind of user behaviour has some consequences. Most of the time, the load evolves in a predictable way. This might be useful for the operator when he leaves his desk at night, as he can set up some NQS specific limits for a given type of load. The negative consequence is that sometimes such jobs will eventually fail, and resubmit themselves at a very high rate, merely because users generally do not check the status of their jobs before sending the next.

Thus the NQS system may become overwhelmed by bad requests that may flood the system. Special measures had to be taken, because this situation often cannot be detected soon enough by the operator, and may lead to disaster when the machine is in unattended mode. A procedure automatically detects such a situation, breaks the chain, mails a warning to the offending user, and closes his batch access until the user has corrected his problem. This procedure has been proved to be very useful, and is now likely to run less often, a few times per month.
T3D CONFIGURATION

As for the NQS Y-MP queues, it was decided that the T3D NQS queues should just continue the standard T3D interactive users limits. Thus NQS queues have been initially set up for 16, 32, 64, 128 and 256 PEs, for times ranging from 300 up to 3600 seconds. Later, it was decided to also create a few queues with longer times, that would only be opened at night and during the weekend.

For Y-MP work, the PATP members can use the T3D NQS queues by just setting the number of PEs to zero. For the same amount of work, the T3D queues allow to the job to run faster than for the pure Y-MP queues. So the PATP members receive better service.

The T3D is configured with only one pool, for both batch and interactive usage. In the beginning, the T3D load was controlled by a number of queue_run_limits applied on each NQS queue. It turned out very quickly that this method could not support many incoming batch requests, and was not giving good response time for short interactive commands, without constantly juggling with the NQS parameters.

Many small partitions (T3D jobs) were blocking larger jobs, that could not enter because of lack of space for a given partition shape. So we informed the users to specify an express time in their mppexec commands, whose maximum value was set to 600 seconds. This allowed small and short partitions to pass in front of other bigger and blocking partitions. Nevertheless, when the maximum wait time is reached (set to 1800 seconds), partitions are blocked again.

In order to better control job flow, we had to build up some local tools that would display the T3D load and provide some production figures and statistics. Collected information are display in Figure 1. These data are systematically recorded in a database. Abstracts from this data base can be collected at any time, and viewed with graphic tools, for the purpose of comparison between separate NQS configurations and load types.

In the meantime, with the new T3D OS releases, Cray Research provided some new commands. This really helped us in better understanding the underlying partition allocation process, and allowed us to define other scheduling tuning paradigms.

THE T3D PARTITIONS AND ALLOCATOR PROPERTIES

A number of tests were made in order to discover and better understand the T3D partition allocator algorithm. These tests were done by loading the partitions on various physical addresses on the T3D using the -base parameter of the mppexec command for some short programs. The mppview command showed us where the partitions were allocated on the T3D torus.

All the partitions have power of 2 fixed sizes and shapes.

We observed that each allowed partition geometry, as defined by the configuration driver, was set in such a manner that any shape could be divided on two half-sized partitions. In other words, each time a given allocated partition becomes free, among other possible combinations, exactly two other half-sized partitions can be recursively allocated at the same place for all partition sizes. This could have been not allowed, but the predefined set of shapes, as released by Cray, just have that property.

We observed that sometimes a given geometry could not be found, even if a sufficient numbers of PEs are seemingly available (cf. Figure 2). This occurs because of the partition allocating history. Most of the times this is due to a partition with a small number of PEs (8, 16 or 32) which has not finished its work yet, whereas all surrounding partitions have. Should this partition remain in the middle of an area usually allocated for larger jobs, then the other jobs may have to wait. This situation leads users to a number of complaints, because they were not all aware of the blocking mechanism.

Another property of the partition allocator is that it tries to exactly fill up holes on the T3D whenever possible, and to leave the largest space available for other bigger partitions. So the allocator seems to make the best decision when allocating new partitions.

PRODUCTION PRINCIPLES; PSEUDO-POOL CONCEPT

We are now experimenting with several possible NQS scenarios with 3 goals in mind:

1) have the best interactive access for the T3D during the day.
In order to achieve best overall throughput, there must be 2 distinct modes of operation:

- during the day, allow a maximum number of PEs for batch processing, lower than the physical number of available PEs. This is equivalent to defining 2 virtual separate pseudo-pools for batch and interactive, with a flexible limit in between. Batch partitions are allocated within their own global limit. Interactive partitions are first allocated on their own partition, and if needed can overflow into the batch area, should it have enough free PEs. Then any new batch partition would have to eventually wait until the interactive partitions allocated in the batch area terminate. Inter-queue priorities are set up so small partitions would enter first. This scheme allows interactive usage to have better priority over batch access;

- during the night, set the maximum number of PEs for batch at the physical number of available PEs. This will likely privilege batch jobs, and allow minimal possibilities for interactive usage. Inter-queue priority is set up so large jobs start processing before smaller ones (priorities are set in the reverse order).

We have been experimenting with this configuration scheme by fixing the complex mpp_pe_limit to several values, but 224 seems to be the optimal value during the day, and 256 during the night, for our type of load. NQS queue run_limits play little role under these conditions.

We have been using this scheme for a few weeks, and our experience gave the following results:

**During the day:**

- immediate access to the T3D in most cases for interactive partitions;
- batch jobs enter only if there are enough PEs available in their pseudo-pool;
- more interactive partitions can be allocated without waiting by overflowing the NQS space if this space is not completely used by batch partitions;
- many interactive accesses can be honored per day without waiting.

**During the night:**

- overall good throughput figures

**but, during the day:**

- nearly no possibility for large PEs batch jobs, even if they are of very short duration;
- no possibility of control on interactive usage;
- high cost involved because of the permanent interactive service.

**and, during the night:**

- reduced interactive access;
- need to set up a procedure when switching from day to night schedule;
- medium sized jobs may never get processed.

Nevertheless, analysis of daily statistics and reports showed us that during these last four months we improved our production figures from about 50% to more than 80% on the average. We experienced good batch throughput, and an increasing number of short duration interactive partitions.

**Production Scenarios**

The idea is to define a set of batch job combinations, so the partitions would fully map onto the pseudo-pools at any time. Such combinations are labeled by a production level. Each level is defined by the number of combinations the largest partition allows for each given partition size. For instance, during the night, the production levels for 256, 128 and 64 PEs jobs are:

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1*256</td>
</tr>
<tr>
<td>2</td>
<td>2*128</td>
</tr>
<tr>
<td>3</td>
<td>1<em>128, 2</em>64</td>
</tr>
<tr>
<td>4</td>
<td>4*64</td>
</tr>
</tbody>
</table>

Going for example from level 1 to level 2 is fairly easy, as a finishing 256 PEs partition can immediately be replaced by two 128 PEs jobs. Going in the reverse order is more difficult, because most of the time there will be not enough PEs available.

This is particularly the case when switching from the day to...
night schedule, simply because 256 PEs jobs usually are not allowed to run during the day.

Thus we have set up some T3D overbooking techniques, so larger jobs normally not allowed to run at a certain level during the day can be initiated. These streams will then enter the execute queue, and will wait for PEs resources in a first-come first-served basis, until all preceding jobs needing fewer PEs terminate.

This procedure can be initiated manually or automatically at any time, but primarily at schedule switching times. It would also be called when a type of load change is needed, implying an automatic production level change.

**FUTURE TRENDS AND PERSPECTIVES**

Cray Research recently provided the «rollin/rollout» feature. This has some major benefits:

- It is possible to temporarily pull a partition out of the T3D, and then to reallocate it at another location. This will be equivalent of moving a partition from one area to another, so a required geometry would be made available and a higher level job could enter.
- It allows to temporarily «swap out» a given partition, should a very high priority job be present.
- Job checkpointing becomes possible. This will allow us to save user jobs between T3D shutdowns.
- Rollin/rollout may be initiated under the control of the user. This will give him off-line program analysis possibilities for debugging purposes.

**CONCLUSIONS**

The tests we have made on the Unicos/Max OS has proven that the management of the Y-mp/T3D pair at our site is possible, and can give satisfying results in our attempts to provide a good interactive environment, along with some interesting production figures, without overloading the normal and pure YMP production service too much. Nevertheless, managing the T3D with NQS is not a straightforward process. The NQS system is very popular, but offers static scheduling mechanisms only. Much local work must be done, in order to encapsulate NQS in a set of directives.

Our tests also showed that the T3D management requires specific methods, due to its architecture with its own set of principles, which are very specific. They differ very much from those applicable for traditional time sharing systems.

For example, in a traditional PVP machine, even if the shared memory is not fully assigned, it is very easy to keep few CPUs active, thus achieving 100% usage. A few low priority, long and small jobs running in the background will just allow this. To achieve good productivity with a MPP machine, a maximum number of PEs must be assigned, and some non trivial full mapping mechanisms must be setup. And should a single job requiring a few PEs be present, then the production figures may decrease, due to the geometry blocking mechanism.

Although we succeeded in avoiding most of the blocking situations due to unavailable geometries, we expect to improve the overall T3D productivity with the «rollin/rollout» feature. But if we have the tool, some «gang scheduling» strategy and mechanism still have to be set up.

e-mail: chenais@sic.epfl.ch

---

**PARALLEL RAYTRACING ON THE IBM SP2 AND T3D**

*by Igor-Sunday Pandzic, Nadia Magnenat-Thalmann, MIRALab, University of Geneva, CH-Geneva & Michel Roethlisberger, IBM CH-Geneva*

Le lancer de rayons est une méthode largement utilisée pour générer des images réalistes sur un ordinateur, mais il s'agit d'une méthode gourmande en ressources de calcul, surtout pour des images complexes.

Cet article présente un algorithme parallèle de lancer de rayons avec une répartition dynamique de la charge, basé sur un logiciel du domaine public, Rayshade. Les implementations sur IBM SP2 et T3D sont presentees, et ainsi qu'une comparaison des facettes d'implémentation et des performances de ces deux systèmes.

Raytracing is a widely used method for generating realistic-looking images on a computer, but it still requires considerable computing power, especially when rendering complex scenes.

This paper presents a parallel raytracing algorithm with dynamic workload distribution, based on public domain raytracing software Rayshade. The implementation on the IBM SP2 and T3D is presented, with the discussion of the differences between the two machines with respect to the ease of implementation and the performance.
Parallel Raytracing on the IBM SP2 and T3D

Motivation

Raytracing is a widely used method for generating realistic-looking images on a computer. Currently, most of the 3D modelling and animation systems use raytracing for the final image rendering. However, the algorithm is quite computing-intensive and the demand on processing power grows as the scenes to visualise become more and more complex.

MIRALab is a Computer Graphics research group and one of its activities is the production of computer generated films. As our modelling and animation algorithms enable us to produce ever more complex scenes, the rendering becomes a bottleneck for the production. The rendering of a really complex film, as the one mentioned in [1] would take several weeks. In order to be able to produce more complex scenes and increase our production speed we have decided to implement a parallel raytracer on the IBM SP2 and the T3D. The software we used as a base is the public domain raytracing package Rayshade.

Principles of Raytracing

The input to the raytracing algorithm is the scene - the description of 3D objects geometry, together with the definition of objects’ materials, definition of lights and the definition of the camera. The output is the image of the scene as seen by the defined camera. More precisely, the output is the colour of each pixel of the image (the frame).

In order to compute the colour of the pixel $x,y$ a ray is casted from the camera through the frame at the position $x,y$ into the scene and the intersection with the first object is found (if any). Based on the position of the intersection point, the surface normal at this point, the position and colour of lights and the material of the intersected surface, the light intensity and colour at the intersection point is computed. Then, the ray is reflected and/or refracted based on the reflectivity and transparency of the surface and the process is repeated recursively with the reflected/refracted rays, adding the light intensities at all intersection points to get the final colour of the traversed pixel in the frame. The recursive process stops when the light contribution gets below a threshold, or at user-defined depth of recursion.

Parallelization strategy

As explained in the previous section, in the basic raytracing algorithm one ray is casted for each pixel of the frame and all the rays are independent from each other. This makes the basic parallelization strategy quite obvious - distribute a certain number of rays to each processor, i.e. each processor works on one part of the frame. Each processor must have the complete scene description. This might seem trivial at the first glance, but careful analysis shows some serious problems:

- Each ray will take a different amount of processor time to compute, depending on the complexity of the intersections that occur. If a big region of the frame is distributed to each processor, this will lead to load balance problems as illustrated in figure 2. Obviously the processors 0, 1 and 2 process the rays that don’t intersect with the scene, and P3 will do all the work. As it is impossible to predict the complexity of the intersections for a given part of the frame, the only solution is to allocate smaller, scattered portions of the frame to each processor.

- Rayshade software uses an advanced raytracing algorithm using the information from neighbouring rays’ intersections to improve speed. This brings a big improvement in speed, but the rays are not independent anymore. In order to profit from the advanced algorithm, contiguous blocks of frame pixels (i.e. rays) should be processed by each processor. The larger the blocks, the bigger the benefit from the advanced algorithm.

These two requirements are clearly contradictory - allocating small scattered regions to improve load balance vs. allocating big contiguous regions to profit from the advanced algorithm using neighbouring information. In order to find an optimum between the two opposing requirements we have devised a dynamic workload allocation mechanism that will distribute the workload optimally for every scene. The basic strategy is to begin by allocating relatively large portions, but as the job comes to the end to allocate smaller and smaller portions, thus adjusting quite precisely the load balance.

The Algorithm

We will first present the algorithm regardless of the actual implementation on a particular machine. While discussing the algorithm we will accept the notion of global memory, accessible by all the processors, that will hold the final image and the global variables that need to be unique and shared by all the processors. The actual implementation of this global memory will be discussed later.

Because of the structure of the existing Rayshade software we decided to take a line of the frame as the basic distributable unit. So, when a processor gets a job, it is determined by the starting line (counting from the bottom of the frame) and the job size expressed in number of lines.
The global workload counter is crucial for the dynamic workload allocation algorithm. It contains the number of the next line to be processed (initially 0) and the global workload left, i.e. the number of lines left to be processed (initially equal to the number of lines of the frame). The global counter is continuously updated by all the processors as they do the work.

The flowchart (figure 3) represents the algorithm executed on each processor. First the database is loaded and the variables initialised. In each step of the outer loop the processor will access the global counter to determine the job to do. The size of each job is proportional to the workload left to be done and inverse-proportional to the number of processors. In this way we insure that the jobs get smaller as the work proceeds, with the effect of fine-tuning the load balance towards the end of the processing When the size of the job is determined the pointer to the next line to be processed is updated in the global counter, i.e. increased by the size of the job. As the job is being done, at each line the global workload in the global counter is decremented by one.

### The implementation

The algorithm has been implemented on two parallel computers: a 256-node T3D and a 14-node IBM SP2. C programming language was used on both architectures, but with different parallel programming subroutine sets. The Shared Memory model was used on the T3D, while on the SP2 we used MPI.

While most of the algorithm is quite straightforward to implement, the real challenging issue was to implement the «central memory» for the global counter. The demand is that all processors have read/write access to the global counter. At the same time conflicts have to be avoided (simultaneous writing by two processors).

Both machines have a distributed memory architecture, so no central memory exists as-is. The solution is to place the global counter on one of the nodes and let the other nodes access it. The solutions for this access vary with the programming model used.

The Shared Memory model allows asymmetric communication, i.e. node A can read/write the data in the memory of node B, without node B needing to send/receive the data explicitly. This already solves the problem of the access to the global counter - it is put on one processor, shared memory links are established and all nodes can access the global counter for read and write operations. To avoid conflicts, each node has to reserve the write access before writing to the global counter. This is implemented using a semaphore in the «global memory». Before writing, the node checks the state of the semaphore. If it is free, it puts its ID number into the semaphore, reserving the write access to the global counter. However, since the reading and writing of the semaphore is not instantaneous, there is still a small risk of a conflict: if two nodes access the semaphore at the same time, both will see it as free and both will send their ID to the semaphore. Obviously one ID will be overwritten by the other. To eliminate this risk, the node will wait a short time after writing the ID to the semaphore, then read the semaphore again to check if its ID is there. Only then it is completely safe to write into the global counter.

In the MPI programming model there is no possibility to access another node’s memory directly. If node A wants data from node B, B has to send it either explicitly or through a global communication scheme. In any case, for each communication a more or less explicit communication pair is needed. There is however the possibility to have non-blocking send and receive commands. Practically, if node A knows that node B will send a message, it can issue a non-blocking receive command, continue its work and from time to time check if the message has been received.
As it is impossible to implement a passive «central memory», we have used the notion of the master processor, whose task is to keep the global counter and distribute the jobs to other processors. However, this task is relatively small compared to the real processing, so it would be very inefficient to waste one processor just to act as the master. For this reason we have implemented an interrupt mechanism where the master processor actually does a slave's work most of the time and gets interrupted each 20 ms to perform the function of the master.

The master will issue non-blocking receive commands allowing all slaves to send him messages asking for a job. When the master receives such message, it will allocate the job according to the state of the global counter and send the job command to the slave. This is a slight change of the original algorithm where the slave performs the job allocation itself. As the slave processes each line it will send messages (non-blocking) to the master to keep him up to date about the state of the work so that the master can update the global counter.

**Performance results**

Computation has been undertaken for the T3D and for the IBM SP2 on scenes of different complexity. A scene of 18000 polygons has been computed once at normal image size 720 x 576 (scene 1) and once at a larger image size 1000 x 1000 (scene 3). An additional scene of higher complexity, 56000 polygons (scene 2) has also been computed on the SP2. The performance results obtained are presented in Figure 4 where we represent measured elapsed time and measured efficiency on both parallel systems. When running on n nodes, the efficiency is defined as \( \text{Eff} = \frac{T_1}{T_n \times n} \), where \( T_1 \) is the time when run sequentially on 1 node and \( T_n \) the time on n nodes.

**Following comments can be drawn:**

1) SP2 processor is about 3 time faster than the T3D processor on this application.

2) T3D exhibits about 4% better efficiency than the SP2 for scenes of modest complexity. This gap tends to reduce for higher scene complexity. Unfortunately, it has not been possible to run case 3 with 56000 polygons on the T3D, as not enough memory was currently available. 2 factors may contribute to efficiency: 1) load balancing and 2) the factor \( f = \alpha \times \beta \), with \( \alpha = \) (speed of communication) / (speed of computation) and \( \beta = \) (amount of computation) / (amount of communication). As we just saw, the SP2 processor is faster than the T3D and the T3D has a faster communication system than the SP2, especially when we use the Shared memory model. Moreover, in the MPI implementation, we introduced an «application latency» of the order of 10ms on the master interrupt mechanism. For all these reasons, \( \alpha \) is higher on the T3D than on the SP2, which explains the differences we measured in efficiencies. On the other hand, computing scenes with higher complexity increases \( \beta \) and therefore have a positive effect on efficiency. We believe that load balancing performs equally good in both systems with the slight exception master node also has to do some job management tasks in the MPI implementation.

**Scalability**

As the application still relies on Host-node model, it was not expected to get full scalability both in term of computing time and data distribution. However, the T3D demonstrates good efficiencies even for high number of nodes. As our SP2 was a 14 parallel nodes system, it has not been possible to run similar tests.

Figure on the front cover page shows a rendered image of the complex test scene.

**Conclusion**

- Both systems have proven to be very efficient on parallel Ray-tracer. Film production time can now be reduced by one order of magnitude by using a reasonable number of nodes, typically 16. This provide a definite advantage of using parallel raytracer in a production environment.
- Our load balancing mechanism proved to be very efficient on both machines.
- The implementation was slightly easier on the T3D with the proprietary shared memory model than in the SP2 with MPI.
- The T3D has demonstrated a faster communication system even for a high number of nodes. However as our application did not require a lot of communications, global performance is higher on the SP2 which can take full advantage of its high performance node. This conclusion might change for a very fine grained parallel application.
- It was possible to compute very complex scenes on the SP2 as more memory was available per node.
ACKNOWLEDGEMENTS

We would like to thank B. Mathews / University of Geneva for kindly making available the whole SP2 for our tests, as well as Jean-Claude Moussal, MIRALab, University of Geneva for creating the final rendered image. Our thanks go also to the EPFL-PATP project whose T3D was used for development and testing.

REFERENCES


T3D EXPERIMENTS OF AN AUTO-TASKED C90 CODE

BY BERNADETTE THOMAS, EDF/DER, F-CLAMART & CHARLES ROGE, TEUCHOS EXPLOITATION - ETABLISSEMENT DE VERSAILLES, F-VERSAILLES

Pour nos besoins en simulation temps réel, la parallélisation sur machines massivement parallèles (à mémoire distribuée) est une solution permettant de réduire le temps effectif de calcul. Partant d’un code auto-tasking tournant sur C90 nous nous demandons comment le migrer sur le T3D. Cela nous conduit au problème plus général du portage d’un code auto-tasking sur une machine massivement parallèle.

Pour illustrer les difficultés d’un tel portage, nous avons procédé à quelques expériences dans le cas de la parallélisation d’un produit matrice pleine-vecteur en utilisant le même nombre de processeurs sur le T3D et sur le C90. Nous voulons que les points suivants soient respectés: les modifications au code source original doivent être mineures et le facteur d’accélération, par rapport à la version séquentielle, doit être attractif.

Quatre approches sont présentées par ordre croissant du nombre de modifications au source. Pour chaque méthode, nous expliquons l’implantation sur le T3D et donnons les résultats qui ont été obtenus.

For our needs in real time simulation, the parallelization on distributed memory massively parallel computers appears to be a solution that allows to decrease the elapsed computing time. Starting from an auto-tasked C90 code, we wonder how to migrate it to a T3D. This leads us to a more general problem, the migration of an auto-tasked code to a massively parallel computer.

Four approaches are presented in increasing order of the number of source code modifications. For each method, the implementation on the T3D is described and the results obtained are given and analyzed.

INTRODUCTION

The development of scientific computing in the past few years shows an increasing use of distributed memory massively parallel computers instead of shared memory vector computers. MPPs form a new generation of computers offering great computational abilities and large memory capacities.

For many years EDF has had in house vector multiprocessors, and has begun to have parallel versions of its proprietary codes. Now, to prepare the future, we are interested in the possibilities of migrating an auto-tasked code, that was parallelized on a C90, to a T3D.

PRESENTATION OF THE PROBLEM

QUICK OVERVIEW OF THE C90 AND OF AUTO-TASKING

The C90 is a shared memory computer.

The processors are linked to a global shared memory which they can access at equal speed. The memory is divided into several memory banks and processors can read or write simultaneously if there is no contention to the same memory bank. An auto-tasked code is composed of sequential regions followed by parallel regions:

Sequential region

parallel raytracing on the IBM SP2 and T3D

parallel raytracing on the IBM SP2 and T3D
Parallel region
Sequential region

In a parallel region data is either shared or private. Any data declared as shared can be accessed by any processor (i.e. parallel task).

On the opposite, data declared as private is private to a parallel task. Each parallel task has its own copy of the variable and any private data cannot be fetched outside the parallel section.

The procedure that consists in determining the scope of the variables is called data-scoping. When data-scoping is done, the work is distributed among the processors via compiler directives (for instance cmic$ do parallel).

Quick overview of the T3D and how to use it

The T3D is a distributed memory computer.

Each processor has its own local memory and is connected to the other processors through a network (a 3D Torus for the T3D).

In all cases the processors run the same code (but not necessarily the same instructions), but there are 2 possibilities to address data: using local addresses or using global addresses.

Local addresses (message passing programming)

Each processor computes its own instructions and can only access data that is stored in its local memory. If a processor needs data that is stored in the local memory of a remote processor, the user must use functions (issued from a communication library), that allow a processor to send or receive data through the communication network. There are 2 communication libraries available on the T3D: PVM, and SHMEM.

Global addresses (CRAFT77 directives programming)

Data is either shared or private. By default, any data is considered as private by the compiler. Each processor has its own copy of the data which is duplicated in all the local memories.

Data can be declared as shared by using a cmic$ shared directive. In that case, the data can be accessed by any processor.

For instance the following directive applied to the array a(4)

\[
\text{cmic$ shared a(i:block(1))}
\]

means that \(a(i)\) will be held in the local memory of processor \(i-1\). If processor \(i\) needs \(a(i+2)\) there will be an implicit communication done by the T3D.

Efficient programming on a distributed memory computer

Definition of speed-up and efficiency

We define the speed-up \(S(N)\) on \(N\) processors as

\[
S(N) = \frac{T_{\text{seq}}}{T_{\text{par}}(N)}
\]

where \(T_{\text{seq}}\) is the execution time of the sequential code on one processor and \(T_{\text{par}}(N)\) the execution time of the parallelized code on \(N\) processors.

\(T_{\text{comm}}\) is the computation time and \(T_{\text{com}}\) is the inter-processor communication time. The parallel efficiency is then defined as

\[
E(N) = \frac{S(N)}{N}
\]

Data locality

In order to decrease \(T_{\text{comm}}\) we have to minimize data transfers between processors. It implies that most of the data which have to be accessed by a given processor should be stored in its local memory. In that case we say that the parallel code has good data locality.

Load Balancing

\(T_{\text{comp}}\) is the computation time of the most time consuming parallel task. In order to decrease \(T_{\text{comp}}\) the work must be shared among the processors as evenly as possible. For instance, if you want to compute 100 parallel iterations (each iteration needing the same computation time) on 2 processors, to get a good load balancing you should execute 50 iterations per processor.

To get a good parallel efficiency on a distributed memory computer, we have to take data locality and load balancing into account.

Main differences between an auto-tasked code and a CRAFT77 code

Execution mode

By default on the C90, sequential sections of a code are executed on one processor and we have to insert directives to obtain a parallel execution.

It is exactly the contrary on the T3D where each processor executes sequential sections by default. If we want a section to be executed by only one processor we must insert the CRAFT77 directives:

\[
\text{cmic$ parallel shared(u,v,A) private (i,j)}
\]

\[
\text{cmic$ parallel do parallel !parallel region}
\]

\[
\text{do i =1,200}
\]

\[
\text{v(i) = 0}
\]

\[
\text{do j=1,200}
\]

\[
\text{v(i) = v(i) + A(i,j)*u(j)}
\]

\[
\text{enddo}
\]

\[
\text{cmic$ end parallel}
\]

A test case

To illustrate the difficulties of porting an auto-tasked code to the T3D, we made some experiments with the parallelization of a full matrix-vector product \(v(n) = A(n,n) * u(n)\). We parallelized the following algorithm on 4 processors either on the C90 or on the T3D (\(n=200\) here):

\[
\text{Compute } u(200) \text{ and } A(200,200) \text{ !sequential section}
\]

\[
\text{cmic$ parallel shared(u,v,A) private (i,j)}
\]

\[
\text{cmic$ parallel do parallel !parallel region}
\]

\[
\text{do i =1,200}
\]

\[
\text{v(i) = 0}
\]

\[
\text{do j=1,200}
\]

\[
\text{v(i) = v(i) + A(i,j)*u(j)}
\]

\[
\text{enddo}
\]

\[
\text{cmic$ end parallel}
\]
do i=1,200  
v(i)= v(i)+2*v(i-1)  \!sequential section  
enddo

Atexpert (performance evaluation tool on the C90) gave us a speed-up of 3.7 on 4 processors of the C90 for the full matrix-vector product.

For the porting, we want the following points to be respected:

- minor source code modifications,
- speed-up compared to the sequential version has to be attractive.

**Different approaches to the porting**

We present four approaches in increasing order of the number of source code modifications. For each method, we describe the implementation on the T3D, we give and discuss the results obtained.

**First CRAFT approach**

In a first approach we use the same data-scoping as on the C90 and we translate the auto-tasking directives into CRAFT77 directives [1] as closely as possible. We increase the dimensions of the arrays to 256, the lowest power of 2 greater than 200, and we declare the data u(), v() and a() as shared:

- Cdir$ shared a(:block,:block)
- Cdir$ shared u(:block)
- Cdir$ shared v(:block).

This picture shows the distribution of data among the 4 processors

The loop on i is parallelized as follows:

- Cdir$ do shared(i) on v(i).

It means that iteration number i is done by the processor which owns v(i) in its local memory.

We obtained a speed-up of 0.6 with this first approach, hence this parallel version is slower than the sequential version of the code.

If we want to get good performance on the T3D it is impossible to ignore memory architecture. We must respect load balancing (as on the C90) and data locality. The first approach does not care about load balancing and data locality.

Processors 0, 1 and 2 compute 64 iterations and processor 3 only 8 iterations (each processor gets 64 elements of vector v(256)). For a good load balancing we need each processor to compute 50 iterations. As far as data locality is concerned, each processor has to access a lot of data that is located in the memory of a remote processor when it executes its part of the work. Data locality is a new concept compared to shared memory computer programming. On the C90, we have a uniform memory access. This is not true on the T3D where it is more time consuming to get a value from the memory of a remote processor than from local memory. On the T3D, we have to distribute the work so that each processor has mainly to deal with data that is stored in its local memory.

In addition, sequential sections that need to access data that is spread among local memories may be slower in the parallel version. In our example, the computation of the sequential loop (\( v(i) = v(i) + 0.005 \times v(i-1) \)) is 8.2 times slower on 4 processors than on 1 processor without parallelization directives. It is impossible to get good performance for our problem with this first approach.

**Second CRAFT approach**

In a second approach, we take data locality and load balancing into account. The matrix is distributed line by line among the local memories:

- Cdir$ shared a(:block(1),:)
- Cdir$ shared u(:block(1))
- Cdir$ shared v(:block(1)).

The work is distributed so that each processor computes all the elements of v() that are located in its memory, and to avoid communications during calculations, the read only shared vector u() is copied into a private vector u_priv(). When a processor computes an element of v() it gets all the data that it needs in its local memory: the corresponding line of the matrix and vector u_priv() (it just has to compute the dot product of the line by vector u_priv()).

The distribution by block of size 1 means that processor i gets the lines of the matrix with numbers equal to i+1 mod[4]: so each processor gets 50 lines of the full matrix. Therefore, load balancing is correct: each processor does the same amount of work by computing 50 dot products.

With this second approach we obtained a speed-up of 3.2 for the full matrix-vector product.

However, some problems remain. Those problems are due to a lack of maturity of the CRAFT77 compiler: the size of arrays must be a power of 2 and, above all, the shared data cannot use the data cache.

In addition, the sequential section is 8.1 slower on 4 processors.
compared to the sequential version on 1 processor without cdir$ shared directives.

Therefore, the CRAFT approach cannot lead to good parallel efficiency of the migrated code. However, it will be interesting to try this approach again with the next release of the compiler: CRAFT90.

**THE DUPLICATION METHOD (SPMD WITH MESSAGE PASSING)**

Our third approach is based upon the SPMD model with message passing. We assume that the memory of each processor can hold all the shared data that have to be used in the sequential section.

**Master slave strategy**

The sequential sections are performed by a master processor, the other processors (the slaves) only work when the master enters a parallel region. The slave processors need input data to be able to perform their computations, this data is sent by the master which scatters the input data.

After the execution of the parallel section, each processor sends the results of its computation to the master processor (data gathering) so that the master is able to compute the next sequential section.

**Data globalization**

The whole code is duplicated on each processor, the sequential regions are performed simultaneously on all processors (redundant work).

With this strategy, we avoid the input data flow and so we minimize the modifications to the original source code. On the other hand the data flow is more important at the end of the parallel computation than with the first strategy. However, using the formula in [4] we noticed that the ratio

$$\frac{T_{\text{gather}}}{T_{\text{globalization}}}$$

is at most equal to 2 and is approximately equal to 1 on 4 processors (for a 2D torus). So we choose the second strategy. To understand the globalization the following example is interesting

```
initialize a()
```

```
do i=1,4
  if (i=1) then a(1)=1; a(2)=2; a(5)=5
  if (i=2) then a(3)=4; a(4)=4
  if (i=3) then a(6)=6
  if (i=4) then a(7)=7
endo.
```

The aim of the globalization is to put the whole array a() in all processors. For instance:

```
1  2  3  4  5  6  7
p0  p1  p2  p3
```

Application to the full matrix-vector product

Each processor gets all the data in its local memory (u(200), v(200) and A(200,200)) before computing the matrix-vector product.

In the parallel region we evenly distribute the work among the processors: 50 iterations per processor.

```
do i=me*50+1,(me+1)*50 !me is the processor number
  v(i)=0
  do j=1,200
    v(i)=v(i)+a(i,j)*u(j)
  enddo
endo.
```

Each processor updates his local memory by computing 50 elements of vector v(). Therefore, it is necessary to perform a communication phase after the parallel region execution in order to put the whole data in each local memory so that each processor is able to compute the following sequential region. This communication phase is called globalization.

There is no need for communication at the beginning of a parallel region. But at the end, we need again to put the results of the computations in each local memory so that each processor is able to compute the following sequential region. This communication phase is called globalization.

There are some limitations to this approach.

The efficiency of this method depends on the migrated program. If the globalization time is slower than the execution time of the parallel region, then the duplication method will not yield good performance.

The communication cost can be too high to get a speed-up (all the data written in a parallel region must be transferred). Moreover,
the redundant execution of the sequential parts involves redundant I/O, which can cause a bottleneck. And in addition, the ported code needs to be held in local memory. This last point is very restrictive.

However, we got the best performance with the duplication method. We can note that the poor efficiency of both CRAFT approaches seems to be essentially the consequence of the non-utilization of the data cache.

(*)cp = clock period

On one processor with the second CRAFT approach, we use a degenerated distribution (the compiler knows that a line of the matrix is entirely in one local memory) and it seems that the CRAFT77 compiler is then able to use the data cache because the execution time is almost as good as on one processor without parallelization directives.

A realistic application

We applied the duplication method to a part of a real thermohydraulics code (the auto-tasked parallelization of this part gives a speed-up of 3.5 on 4 processors of the C90). We were able to use this method because the whole code could be held in a local memory.

There were only 3 shared commons written in the parallel region, as looking for the written shared variables in the C90 data-scoping shows it. For instance

```
COMMON /GRI/TAB1(160),TAB2(160),...
```

and the arrays are written as follows

```
The globalization time could be approximated by the time needed for the globalization of an array composed of 260 parts of 40 elements.

Firstly, we tried to implement the globalization using PVM and obtained a communication cost of 36 ms.

Each parallel task needs about 10 ms of computation time (the sequential execution takes 40 ms), so a communication time of 36 ms implies an elapsed parallel computation time of 46 ms. This elapsed time is too important to get a good speed-up, it is more important than the sequential execution time.

So we tried to implement the globalization by using the SHMEM library and obtained a communication time of 6.5 ms which made us hope to get a speed-up of 2.4 (we predict a parallel elapsed time of 10 + 6.5 = 16.5 ms and so we can expect a speed-up of 40/16.5=2.4)

So we implemented the globalization in the code.

Firstly we had to compute the arrays index() which allow each processor to know which elements of the shared written arrays it actually writes.

```
Cparshmem
Common GRI 260 parts
C Remplissage de index1()
index1(i)=index1(i)+me*40
ilong1=40
do i=2,240
   index1(i)=index1(i-1)+163
endo
Then the globalization phase was implemented as follows

C Common GRI globalization
do i=1,3
   do k=1,240
      call shmem_put(TAB1(index1(k)),TAB1(index-1(k)),
      & ilong1,ivois(i))
   enddo
endo
```

We also had to use the shmem function ucdflush() to invalidate the cache in order to prevent cache coherency problems (we can notice that the modifications on the original code are not very important).

We obtained a speed-up of 2.2 on 4 processors of the T3D. That is good enough but the speed-up was better on the C90.

ANOTHER POSSIBILITY

There is a fourth possibility if the auto-tasked source code has the same distribution of data from one parallel region to the next, and if the parallel regions do not access a lot of shared data. For instance, that is true when the code is based on a subdomain decomposition method. Then, the auto-tasked code can be ported with a good efficiency using a SPMD model and message passing.

To use the fourth possibility, it may be necessary to modify the data structure so that each parallel task accesses «different» and contiguous sections of the memory. Then, data is ready to be distributed among the local memories. In addition, because processors do not use a lot of shared data, the communication cost should not be very important and it will be possible to obtain a good parallel efficiency.

That was done for some EDF codes (for instance see [2, 3]). With this fourth possibility however, the source modifications are not minor anymore, so it does not respect the point concerning the minimization of source code modifications.

CONCLUSION

If a user wants to develop a new application on the T3D, it is more interesting to use message-passing programming in order to obtain a good parallel efficiency.

For porting an existing auto-tasked code, CRAFT programming may be considered an acceptable solution if the following points are improved:

- better utilization of the data cache by the compiler,
- possibility to use shared arrays whose dimensions are not equal to a power of 2.
- the duplication method can be held in a local memory, the duplication method is another solution.

```
T3D EXPERIMENTS OF AN AUTO-TASKED C90 CODE
```

| T3D Workshop | Nov. 95 | 62 |
T3D EXPERIMENTS OF AN AUTO-TASKED C90 CODE

ACKNOWLEDGEMENTS

We thank F. Ifénecker (EDF/SEPTEN) for his help and advice.

REFERENCES


FRENCHESS: A T3D AT THE 8TH WORLD COMPUTER CHESS CHAMPIONSHIP

by Marc-François Baudot, XL, F-Verrières-le-Buisson, Jean-Christophe Weill, Université de Paris 8, F-Cachan and Jean-Luc Seret & Michel Gondran, Direction des Etudes et Recherches, EDF, F-Clamart

This paper presents our work on parallel search algorithms at the University of Paris 8 and the Direction des Etudes et Recherches (E.D.F.).

It focuses on our parallel chess program Frenchess which, running on the Commissariat à l’Energie Atomique’s 128 processors T3D, just finished third of a field of 24 at the 8th World Computer Chess Championship in Hong Kong (May 25 - May 30 1995).

Frenchess relies on the new ADBADA (Alpha Beta Distribué Avec Droit d’Aînesse) algorithm, an efficient and easy to implement parallel version of the alpha-beta algorithm (which was formerly known not to parallelize well). ADBADA had first been tested on a TMC CM5 and a Cray CS6400, and the adaptation to the T3D, using SHMEM, has proved very satisfying.

Even though single PE performance was disappointing, the T3D gave us excellent parallel performance, proving that our ABDADA algorithm scales very well up to at least 128 processors (with a speedup of over 70 in tournament conditions), and that it takes full advantage of the T3D’s fast parallel architecture: it enabled us to build a world class chess program in just a few months’ work. The ABDADA algorithm can certainly be useful for the parallelization of related problems, such as Branch and Bound problems, on this architecture.

INTRODUCTION

The Frenchess program is a joint research project on parallel search algorithms between the Direction des Etudes et Recherches (D.E.R.) of Electricité De France (E.D.F.) and the Paris 8 University, where Jean-Christophe Weill obtained his PhD Thesis on the subject of computer Chess1 [16].

Computer chess is not only a matter of prestige: it is a good test-bed for parallel search techniques because search speed is of paramount importance in chess programs, and international competitions permit to judge the achievements of the contestants on a variety of hardware. Competitions are a very good motivation for the different teams working in this field to get the most out of their machines. The net result of the Frenchess project, up to now, is a new parallelization technique, ABDADA (which we will describe in the second section), that gives excellent performance for this class of problems similar to Branch and Bound, on this type of architecture.

1 Jean-Christophe Weill and Marc-François Baudot have written several chess programs in the past few years. They have obtained the titles of Personal Computer Chess Champion and Amateur Micro-Computer Chess Champion in 1990 with the program Echec 1.9, and Amateur world Champion with Cumulus 2 in 1992.

2 IBM’s Deep Blue project is clearly aiming at the human world champion, Garri Kasparov. Thus one of our goals was to catch up with Deep Blue and then to beat them in the race to produce the best chess playing entity in the world. The results of the Hong Kong world championship show that it took us little time to nearly accomplish the former objective.
FRENCHESS: A T3D AT THE 8th WORLD COMPUTER CHESS CHAMPIONSHIP

The availability of these machines, except for those that rely on special hardware (but then, even though the machine might be available 24 hours a day, it’s only one machine...) is very bad, as they are usually heavily used for other tasks. Debugging and fine tuning the program cannot be done on slower computers, as search extensions and evaluations behave differently depending on the speed of the computer1. The program has to be specially written for machines which evolve faster than the PC architecture.

Parallelizing the αβ algorithm is, in itself, a difficult problem [16][12][13][4][5].

One of the most famous PC programs, Chess Genius, did not do very well and finished only sixth. Less known to the public, the parallel programs were:

- **StarSocrates**, developed at the MIT Laboratory for Computer Science, runs on a 1824 nodes Intel Paragon. Each node consists of two 50MHz 1860 processors. It uses a work-stealing based search [11][12] and finished second. They claim a speedup of 300 to 400 for 3648 processors. Despite this, their games were not impressive, with the exception of their good win over the PC program Rebel. For instance, their last round win against Hitech was only due to Hitech’s aberrant endgame play.

- **Deep Blue Prototype**, from IBM, uses 14 dedicated chess search processors in parallel, each processor searching 400000 to 500000 positions per second. It was the clear favourite, but stumbled on a position which was objectively superior but very difficult to understand for a program2. Then it lost a game to the PC program Fritz (which eventually won the tournament), due to a bad hole in the opening book. The other 3 games it played were all very impressive. Deep Blue finished equal third with Frenchess (searches around 500000 positions per second on the 128 processors T3D).

- **Zugzwang**, using the YBWC algorithm [6][4][3] on a 192 processors GC-Powerplus. Zugzwang tied for 6th place. They were disappointed with single processor speed (just like we were), but it seems their communication speed could not make up for that, so we feel we were much better of with the 128 processors T3D. Even though Zugzwang is now an old program, they had a nasty bug in the first rounds.

- **Phoenix 89**, using a 20 processor Sparc 2000 server and the DPVS algorithm [14], finished equal 14th.

- **Frenchess**, on the 128 PEs T3D exhibited a high level of play, but due to bad connection problems (which plagued all the remote machines), could not use its special tournament opening books3. It had to use a book designed to play against humans with risky variations, and lost a game to Chess Genius due to a bad opening line. The other games were more satisfying, and in the last round it crushed its DEC Alpha based opponent with a move that surprised its opponent and the IM and IGM spectators! Shortly after, when the game was over, indicating the move was a brilliant sacrifice and not a blunder, it was called the best move of the tournament by the arbiter. The algorithm parallelized is the αβ algorithm, but the new ideas

---

1 Games are played in a limited time, so the faster the computer, the greater the search depth. The best way to use the computing power is not the same if you are using a PC or a CRAY: the most useful work is not the same at every depth. Simulating the behaviour of a CRAY on a PC would just take years to give enough useful results!

2 This is the case of most heterogeneous material position: keeping the right pieces is often a difficult decision based on long term planning and pattern recognition. Deep Blue made the wrong decision after a well played game from both sides, and could only draw against the strong PC program WChess.

3 ftp was just not working from Hong Kong. The reason the book had to be sent by ftp is that our opening book tools are PC programs. The opening book is specially designed against specific opponents, and next round’s opponent is only known a day before the game.
which have been introduced should also be of great interest for related subjects such as branch and bound problems. The most important of those ideas, in the Frenchess project, is the ABDADA algorithm defined by Weill, which gives very good speedups (which were thought impossible just a few years ago) and is very easy to implement.

**THE ABDADA PARALLEL MINIMAX ALGORITHM**

The ABDADA search algorithm introduced in [16] is a parallel version of the αβ search algorithm[10]. For the reader interested in understanding the details of the ABDADA algorithm, we recommend first to become familiar with the αβ search algorithm, iterative deepening and transposition tables which are well known and important parts (but clearly not the only ones) of what make chess programs play so well [14] [16]. However this is not necessary to understand the basic ideas.

ABDADA is based upon both the YBWC [7, 5, 6, 4, 3] and the αβ* [2] algorithms. From YBWC, it keeps the basic concept: parallel evaluation of successor positions of a game position is allowed if and only if the eldest brother is fully evaluated. From αβ* it inherits the use of a shared transposition table to control the speculative parallel searches.

ABDADA is a particularly elegant way to parallelize the αβ search since it does not require derecursification of the αβ search, nor any division of search trees either statically or dynamically amongst processors, and no passing of sub-windows. It was proven to be easier and quicker to implement and to debug than related algorithms [16]. It is really close to the sequential algorithm, so all sequential search heuristics can be easily implemented.

One of the most efficient heuristics used in computer games is the transposition table heuristic. In its most simple expression, it consists of a table of the results of positions already explored in the search, or of the positions seen in previous iterations when using iterative deepening. The results stored in the transposition table by a sequential chess program are the depth of the search from that position, the best move from that position, and the score so far.

The ABDADA algorithm can be explained as follows:

1. Let tt be a shared transposition table. To the standard definition of the table we add, for each entry, a new field nproc which is the number of processors currently evaluating the node related to this entry;
2. All the processors begin the search simultaneously at the root of the game tree;
3. When a processor enters the evaluation of a position p, it increments the field tt[p].nproc in the transposition table;
4. When a processor leaves a position (because this position is fully evaluated or has been pruned), it decrements tt[p].nproc.

The analysis of a position is done in three phases:

1. The eldest son is analysed, regardless of the position of the other processors;
2. the other sons which are not currently analysed by other processors are analysed;
3. then analysis of the positions which are not totally evaluated (i.e. the corresponding entry in the transposition table is not set for the analysis depth) is done.

This can also be expressed as shown in the figure 1. ABDADA has the following advantages:

- Easy and quick to implement: the main algorithm is very close to the sequential algorithm.
- Efficient for small and deep problems.
- Failure resistant: a time-out when waiting for transposition table answers ensures that the algorithm will return a correct answer even when only one processor is alive! Its main drawback is the need for a very fast global transposition table. This is where all the implementation effort has to be made. A network of workstations using PVM will not offer sufficient communication speed for ABDADA.
- Performance of the ABDADA algorithm will be discussed in the next section.

**PORTING FRENCHESS TO THE T3D**

From the previous section, it is obvious that the critical part of Frenchess is the access to the global transposition table.

On the CM5, shared memory was simulated, each processor being in charge of a part of the global hash table; we had to rely on active messages to get some very good performances which CMMDD could not give us otherwise.

This method of shared memory simulation has the advantage of making simultaneous accesses to a same entry impossible (this is important to ensure that ABDADA works correctly, i.e. that the number of processors is correct for a given transposition table entry). An average speedup of 16 for 32 processors was obtained on a standard set of test positions known as the Bratko Kopec positions [16]. On the Cray CS6400, shared memory was used, mutex (mutual exclusion) locks guaranteeing exclusive access to each hash table.

1. More precisely the negascout algorithm [16].
2. It should be remembered that its efficiency depends on move ordering: searching the best move first is the goal of many heuristics found in chess programs.
3. Rather than going directly for a depth n search, whose computing time cannot be known beforehand, a chess program first performs a depth 1 search, then a depth 2 search, and so on until its time control algorithm interrupts it. This has two very interesting advantages:
   - Time control is easier: when the search is interrupted during a depth n search, we are sure that all the moves have been searched to at least depth n-1;
   - Information is stored during depth n-1 search that will permit depth n search to order the moves efficiently. Iterative deepening is faster than searching directly at depth n.
4. Information (i.e. score, depth of search and best move from that position) acquired during a depth i search are saved in a hash table. If a position is met twice through different paths, the second time its subtree will not be searched. Depth i+1 search, when reaching a position already searched at depth i, will first look at the move stored in the table, ensuring a better ordering of the search tree.
5. Transposition tables are often called familiarly hash tables because they make use of a hash key.
6. In Chess, it is not uncommon to reach twice the same position (i.e. the same pieces on the same squares with the same player to move) by different paths.
7. ABDADA is the acronym of the french ‘»Alpha-Bêta Distribué avec Droit d’Aînesse» which can be translated as «Distributed Alpha-Beta Search with Eldest’Son Right» («droit d’aînesse» meaning birthright).
entry. Precise measurement of the speedup was not possible on this machine but it looked very much like what we had seen on the CM5 even though single processor speed was around four times greater.

When we had to adapt Frenchess to the T3D, we did not find anything to replace mutex locks in MPP C! Such locks existed in CRAFT or on other CRAY systems (c_lock), but were not supported on MPP architectures. So we first decided not to use SHMEM and resort to message passing until we found a solution to this problem. Another reason why you might not want to use SHMEM is that, according to the documentation, it does not co-operate very well with the cache memory.

So we first tried an all PVM version. On the other machines we had used (including networks of workstations), the ABDADA algorithm was clearly too slow with PVM. On the T3D, PVM is so much faster that ABDADA was able to give some «interesting» (i.e. speedups and not slow-downs!) speedups but still, it would not have been wise to enter a world championship with this!

Then came the time to try setting channels between processors. Unfortunately we never really succeeded in making channels work correctly twice in a row (initialisation problems, coming maybe from a bad understanding of the documentation), and the world championship was not very far away!

So we turned to SHMEM again, and people working for the CEA and a Cray engineer pointed to us that mutex locks could be simulated using the shmem_swap call. Shmem_swap has the interesting property of being an atomic operation (this is what we missed when we were looking for a replacement to mutual exclusion locks).

It should be noted that SHMEM seems to be much more reliable than PVM or channels, especially under the debugger Totalview. We feel that SHMEM documentation[1] should be made less frightening, since it enabled us to get rather easily very high performance.

One last remark: when looking for maximum performance on the T3D, one should refrain from using some calls such as flush, printf or time which generate calls to the front-end. It was observed that we were using a lot of the front-end's (C90) CPU, which seemed rather curious and penalised both the T3D and the C90. We found out that our time control algorithm called time several times per second (on only one PE). When we replaced time with clock, the problem did not exist anymore.

### Parallel performance on the T3D

Each processor stored 1M transposition table entries (five 64 bits words per entry).

We have measured the loss of performance on the sequential speed (measured in positions visited per second) due to the use of SHMEM. Judging from the measures we have done since we returned from the World Championship, this loss does not change noticeably with the number of processors.

The figures in the following table are an average on positions representative of the different phases of the game (opening, middle game and endgame). The time is the total time for the tests in seconds, the speed given is the average speed reached at the end of the tests on a single PE. For each test, the time was the time needed to complete a fixed number of plies, so that each tested version did exactly the same search. All the hash table entries being held on one PE (either the one doing the search or an idle PE), we compared the speed reached when:

1. The processor uses its own memory to store the transposition table, and accesses it normally (no SHMEM).
2. Same as 1 but with the locks enabled (although it will never have to wait for a lock to be released since it is the only active processor).
3. The processor accesses its own memory using SHMEM, and locks are enabled.
4. The processor accesses another processor's memory (in a partition of 2 processors) with SHMEM and locks are enabled.

<table>
<thead>
<tr>
<th>Case</th>
<th>Time(in seconds)</th>
<th>Visited Positions per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1198</td>
<td>5071</td>
</tr>
<tr>
<td>2</td>
<td>1218</td>
<td>5001</td>
</tr>
<tr>
<td>3</td>
<td>1260</td>
<td>4825</td>
</tr>
<tr>
<td>4</td>
<td>1288</td>
<td>4778</td>
</tr>
</tbody>
</table>

Of course this would be meaningless without indicating the number of SHMEM calls. In these positions, the average number of shmem_swap per second was between 2600 and 3200, shmem_get ranging from 1300 to 1600 and shmem_put were in the range 950 to 1200 per second.

On the same positions, searched to the same depths with 128 processors, we get 422000 to 855000 shmem_swap per second, 166000 to 253000 shmem_get per second and 81500 to 88000 shmem_put per second for middle game positions, and as much as 1124000 shmem_swap per second for only 41000 shmem_put/ and 118000 shmem_get/s for a pawn endgame position which, using only one processor, did not single itself out particularly.

One should note that these figures are different from the number of «loads» and «stores» of hash table entries. Just searching for a hash table entry may require several shmem_swap (because an entry is already locked or because of the collision handling strategy, cf. [16]) and shmem_get, and just «loading» from the hash tables will also require calls to shmem_put. With no collision and no concurrent access to a same entry the number of shmem_swaps should be exactly double that of shmem_get/s (one for unlocking, one for unlocking), which is what we observe for one PE.

When the two PEs (the one doing the search and the one holding the transposition table) are any two PEs in a bigger partition (but the only active ones), we have not been able to measure any performance difference. Of course, this does not imply that SHMEM communication speed between any two PEs in the T3D is exactly the same, but that, as stated in the documentation, communication speed in the T3D is very little dependent on distance. Trying to improve performance using partition shape, or, in the case were only 128 processors were in the range 950 to 1200 per second.

3 These blocked pawn endgames, where transposition tables dramatically improve the speed of the sequential search, are somewhat of a problem for parallel searches. At depth 13, the speedup for the 128 processors was only 17 while it was searching 200 times more nodes per second than the 1 processor version.

---

1 The only way would have been to make sure that nobody else used the same partition as Frenchess. We did not try to negotiate this as we knew we would be moving to the T3D soon and the CS6400 was rather heavily used.

2 which are rather shallow for 128 processors.

3 These blocked pawn endgames, where transposition tables dramatically improve the speed of the sequential search, are somewhat of a problem for parallel searches. At depth 13, the speedup for the 128 processors was only 17 while it was searching 200 times more nodes per second than the 1 processor version.
some of the processors would hold parts of the transposition table, with the distribution of these processors, would most probably have been a waste of time. Therefore we think that it is easier to get the best parallel performance possible on the T3D than on the CM5.

Cases 1 and 2 show that the calls to shmem_swap are almost not penalising (but with more than one processor they might imply waiting for locks to be released). Cases 3 and 4 compared to cases 1 and 2 show that the use of SHMEM gives roughly a 5% penalty, which is, in this context, a very good result.

Of course, when using more than one PE, things might be different, but the fact that the number of visited positions per second on a PE remain fairly stable when the number of PEs increases tends to show that SHMEM still performs very well and that locks are not a performance problem. This seems confirmed by the behaviour of the program at the World Championship, but precise measurements of parallel performance require large sets of positions and long search times, and these measures are not yet completed.

So the T3D seemed well suited for the ABDADA algorithm which proved very efficient on the 32 nodes CM5, but would the ABDADA algorithm still yield good speedups with 128 processors? We have not had time to go through all the measures done on the CM5. In fact, it would have been much longer to get significant results on the T3D since it has more numerous and faster processors: a problem of reasonable size on 128 nodes would become really long on one node, and we have to make enough measures to get a meaningful average.

We have used the 24 Bratko Kopec positions, searched to a fixed depth. We do not think that this set of positions is satisfying, but they are used by most other authors and give a basis for comparison. We have also used other test positions to verify our results.

We have first adopted the practical chess programmer’s point of view. We have not measured speedups with a constant transposition table size: when using 128 processors the transposition table of view. We have not measured speedups with a constant hash table size have been measured with 128 processors compared to 32:

<table>
<thead>
<tr>
<th>depth</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>speedup</td>
<td>1.05</td>
<td>1.13</td>
<td>1.23</td>
<td>1.22</td>
<td>1.49</td>
<td>1.98</td>
<td>2.62</td>
<td>2.68</td>
</tr>
</tbody>
</table>

Relative speedup of 128 processors compared to 32 with a constant transposition table size

During a real game, where each player has 2 hours to play 40 moves, Frenchess searched most positions to depths 10, 11 or 12 (much more in simplified endgames). A conservative estimate for depth 10 speedup, in the same conditions as above, would be around 75 for 128 PEs. Thus we feel ABDADA gave us excellent performance on the 128 processors T3D.

**Single PE performance**

This was our big disappointment with the T3D.

Since the version of Frenchess on the CM5 was used for measuring performance, we kept it very stable over a long period of time. This was not possible with the CS6400 and T3D versions which were designed to obtain maximum performance from a chess point of view and had to be ready at very short notice (we started working on the T3D in February 1995 and the World Championship was in May 1995). Improvements in the evaluation function and to heuristics such as move ordering made each new version of the program difficult to compare with the last one from the point of view which interests us here.

Nevertheless, we have always had the possibility to switch back to the very simple evaluation and search extensions used on the CM5 version. This enabled us to make meaningful comparisons on the sequential speed of each version of Frenchess. The unit in which we measure the speed of Frenchess so as to be able to compare its performance on different architectures is the number of positions visited per second. Here we give an estimate of the speed on one PE of Frenchess using a fast evaluation and simple extensions, which explains the differences between these figures and those given above.

- On a 50 MHz Sparc 10 (or a single processor of the Cray CS6400), the sequential algorithm was visiting around 20000-25000 positions/second. This was also the speed we obtained on 150 MHz DEC Alpha stations.
- When using shared memory for the transposition tables on the CS6400, this speed went down to around 12000-15000 nodes/second (we are not stating here that this decrease in speed is

1. On the CM5, where shared memory was simulated by message passing using active messages, it was advantageous to dedicate some processors to the transposition table storage and retrieval while the others took care of the search.
2. We would have had to use the whole machine for several hours just to measure the performance of one processor with as much memory as with 128 PEs. Doing it the other way around does not make much sense. Our problem (the world Championship) being a short term practical one, we spent machine time in a more useful way. For the same reason, we have not measured depth 10 searches with only one PE.
due only to shared memory accesses since the parallel hash table code is slightly more complex, involving mutex locks and more information than in the sequential version).

We expected to get higher figures on the T3D, but the speed on one PE was in the 7000-9000 positions/second range!

On the other hand, we have seen that we were not losing much speed from the communications which makes up for these bad results. We think the poor single PE performance we noticed was primarily due to 2 problems:

1. cache problems (cache too small, cache invalidation?)
2. poor optimisations from the compiler. The only option we found to really improve program speed was... not using the –g (debugging) option!

We know from experience that this sort of program benefits a lot from big cache memory and advanced optimisations from the compiler, but we spent quite some time trying to improve single PE performance with only marginal results on the T3D.

CONCLUSION

We have shown that ABDADA still gets excellent speeds with 128 processors on the T3D, and when comparing the behaviour of the T3D with that of the other machines we have used (32 nodes CM5, CS6400) for this class of parallel algorithms, we have found the T3D to be the easiest to use architecture.

Porting of Frenchess to the T3D only began in February 1995, and in May 1995 Frenchess finished third of the World Championship, which shows that

- ABDADA is well suited to the T3D. If that did not prove to be the case with a bigger number of processors, other (but more complicated to program) recent known methods could be used to get good results.
- the T3D was easy to program with excellent parallel performance using SHMEM once the problem of the mutex lock was solved. It seemed to us that the T3D was an excellent machine as far as communications are concerned, but that its sequential performance could be easily improved by adding more cache memory, improving the compiler and maybe using a faster Alpha processor. It was our understanding, from the documentation, that software was not yet in its final state, so our hopes of sticking to the same architecture and benefiting from better sequential speed seem realistic.

ACKNOWLEDGEMENTS

We wish to thank all the people at the C.E.A., E.D.F. and CRAY who have helped us very efficiently and have made it possible for us to be ready in time for the World Championship, particularly: D. Beaucourt, C. Caremoli, L. Colombet, J.-E Duboisset, J.-M. Fieni, O. Goudard, R. Napoleone, G. Roth, Y. Souffez.

REFERENCES


e-mail: marc-francois.baudot@der.edf.fr, weillj@lifl.fr, jean-luc.seret@der.edf.fr

Or nodes per second. They do not mean anything when comparing two different programs. They also vary very much from one position to the other. Parallel numbers of nodes per second are even more difficult to interpret because of the amount of wasted parallel work.