

## **New Fortran Features:**

### **The Portable Use of Shared Memory Segments**

Ian J. Bush  
CCLRC Daresbury Laboratory,  
Warrington,  
WA4 4AD,  
United Kingdom  
Email: [I.J.Bush@dl.ac.uk](mailto:I.J.Bush@dl.ac.uk)

#### **Abstract**

The latest version of the Fortran standard, Fortran 2003, introduces many new features. While full compilers are not yet available, many have now implemented one of the most important new features, a portable method for Fortran to interoperate with C. This allows access from Fortran to many features of the OS that were previously the domain of the systems programmer. In this report I show how shared memory segments and semaphores can be used to both enable HPC applications to solve bigger problems and to solve those problems more quickly. I illustrate these points with an example using the GAMESS-UK electronic structure code.

**This is a Technical Report from the HPCx Consortium.**

Report available from  
[http://www.hpcx.ac.uk/research/hpc/technical\\_reports/HPCxTR0701.pdf](http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0701.pdf)

© UoE HPCx Ltd 2007

Neither UoE HPCx Ltd nor its members separately accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations.

---

<b>1</b>	<b><i>Introduction</i></b>	<b>3</b>
<b>2</b>	<b><i>Fortran Interoperability With C</i></b>	<b>5</b>
2.1	Introduction	5
2.2	Interoperability of Intrinsic Types	5
2.3	Interoperability with C Types	6
2.4	The Value Attribute	6
2.5	Invoking C Functions from Fortran	7
<b>3</b>	<b><i>System V IPC Facilities</i></b>	<b>7</b>
3.1	Shared Memory Segments	7
3.2	Semaphores	9
<b>4</b>	<b><i>Bringing It All Together: FIPC</i></b>	<b>10</b>
4.1	Contexts	10
4.2	Use of FIPC	11
4.3	Implementation	13
<b>5</b>	<b><i>FIPC in Action</i></b>	<b>13</b>
5.1	GAMESS-UK	13
5.2	Results	15
5.2.1	Nitrobenzene (PhNO <sub>2</sub> )	16
5.2.2	Primitive Cubic Hydrogen (PCH)	18
5.2.3	Isocitrate Lyase	19
<b>6</b>	<b><i>Conclusions</i></b>	<b>21</b>
<b>7</b>	<b><i>Acknowledgements</i></b>	<b>22</b>
	<b><i>References</i></b>	<b>22</b>

## 1 Introduction

The latest version of the Fortran standard, Fortran 2003 [1], introduces many new features. While full compilers are not yet available many have now implemented one of the most important and useful new features: a portable method for Fortran to interoperate with C. These include both commercial compilers, such as those from IBM and Intel, and free ones, such as g95, gfortran and the Sun Fortran compiler. Therefore it is now possible for Fortran to call C routines and access C data structures in a portable way and *vice versa*. This is a vast improvement over previous methods which have required:

1. detailed knowledge of how arguments<sup>i</sup> are passed between the two languages;
2. knowledge of the size of intrinsic data types;
3. knowledge of how derived data types are laid out in memory;
4. knowledge of how symbols for the linker are generated;

all of which depend not only on the operating system (OS), but also on the compiler (for both languages) and the linker.

For the Fortran programmer the new standard allows portable access to a very wide range of software written in C. Since this software is very often **not** numerical in nature this provides a complementary functionality for the Fortran programmer. One good example is OpenGL, one of the standard graphics APIs. Interoperability with C allows direct use of this API (Application Programming Interface), and so a Fortran application may, for instance, portably provide graphical analysis of any results it generates.

A very wide variety of standard APIs, such as OpenGL [2], pthreads [3] and SDL [4], are available by this method. Possibly the most important to a Fortran applications developer are those in the OS, which, whether the OS be a variety of Unix or Windows, are usually defined in terms of C. A simple example is `gethostname`.

```
#include <unistd.h>
#include <stdlib.h>
int gethostname( char *name, size_t len );
```

which returns in `name` the host name of the machine on which the application is running. As this is a POSIX.1-2001 (Portable Operating System Interface) compliant routine it is available on both Unix and Windows. Therefore a Fortran application can use it in combination with C interoperability to obtain this information, something that is difficult to achieve by other means.

In this report I shall focus on the use of the standard System V inter-process communication (IPC) facilities. As these are also POSIX.1-2001 compliant they

---

<sup>i</sup> Fortran usually uses the term “arguments” while C uses “parameters.” As the latter has a different, specific meaning in Fortran I shall use the former throughout to avoid confusion.

provide a portable method for accessing shared memory features. A number of facilities are provided, including methods to

1. create and destroy areas of memory, called segments, that may be shared between processes; and
2. create locks, barriers and critical regions to avoid race conditions when those areas of memory are being accessed.

Many large-scale parallel systems are now built from shared memory SMP (Symmetric Multi-Processing) nodes. On SMP nodes, especially the “fat” (16 processor, 32GByte) SMP nodes that HPCx provides, IPC may allow the application programmer to save large amounts of memory by having only one copy per node of a replicated data structure, rather than one copy per process. For example the definition of a complex physical system on which a calculation is being performed may result in each process having to access a large amount of data. If the system definition is held in a shared memory segment a sizeable saving in memory per node may be achieved on an SMP.

In some ways the purpose of this work may be seen as providing some OpenMP-like facilities for an MPI application. Therefore it is reasonable to ask why these complex, unfamiliar System V IPC methods are proposed, rather than developing the application in mixed mode OpenMP/MPI. There are a number of reasons.

1. There are a very large number of mature MPI applications for which a re-write in mixed mode OpenMP/MPI is simply not practical. GAMESS-UK, at over 800,000 lines, falls into this category.
2. Many codes rely on parallel numerical libraries. While libraries that use either pure MPI or a purely threaded paradigm are common, mixed mode libraries are very rare. Codes which rely on numerical libraries are therefore difficult to cast into a mixed mode form. Again GAMESS-UK is an example.
3. The System V IPC method is more portable; for instance it does not rely on a thread-safe MPI library.
4. The implementation described here is very easy for the application programmer to use with no knowledge of the system calls being required.
5. In the opinion of the author, OpenMP is not necessarily any less complex or less intrusive than the System V IPC method.

The remainder of the report is organized as follows. In the next section I shall introduce the C interoperability features of Fortran 2003, and in section 3 the features of IPC that are required by this work. Section 4 will introduce briefly the Fortran module that has been written, FIPC, to allow easy and efficient access to shared memory segments by Fortran applications. Finally I shall illustrate use of that module with GAMESS-UK, and show how IPC can be used to improve dramatically both the memory usage and performance of a code that requires large, replicated data structures.

## 2 Fortran Interoperability With C

In this section I shall briefly introduce those features of interoperability with C that are relevant to this work. A full description of all the features may be found in [5], which I shall follow closely.

### 2.1 Introduction

The Fortran 2003 standard provides a standardized mechanism for interoperating with C. Methods are provided for a Fortran subprogram to access C data structures and to invoke C functions, and conversely for a C function to access Fortran data structures and to invoke Fortran subprograms. For this to be possible it is clear that any entity that is accessed by both programming languages must have equivalent declarations; there must be a method of mapping the Fortran declaration onto that used in C and *vice versa*. This is enforced within the Fortran subprogram by requiring all such entities to be interoperable, and the standard describes how that may be achieved for data types, variables and procedures.

In this work only a subset of the interoperability features are used. I shall describe in turn interoperability of intrinsic types, interoperability with C types, the `value` attribute and invoking C functions from Fortran.

### 2.2 Interoperability of Intrinsic Types

This is achieved by use of the `kind` mechanism introduced in Fortran 90. A new, *intrinsic* module, `iso_c_binding`, is provided by the language. This contains named integer constants which are the kind values that a Fortran intrinsic type must have to interoperate with a C type. Some of the more important are:

Fortran Type	Kind Value	C Type
Integer	<code>c_int</code>	<code>int</code>
Integer	<code>c_short</code>	<code>short</code>
Integer	<code>c_long</code>	<code>long</code>
Real	<code>c_double</code>	<code>double</code>
Character	<code>c_char</code>	<code>char</code>

Thus to interoperate with a C variable declared as

```
short foo;
```

the Fortran declaration should be

```
Use, Intrinsic :: iso_c_binding
Integer( c_short ) :: foo
```

For the Fortran `Character` type, interoperability requires that the length parameter be 1.

### 2.3 Interoperability with C Types

The dependence of C upon pointers requires that the Fortran standard defines methods to interoperate with them. For this the `iso_c_binding` module also contains a definition of a derived type `c_ptr` which is interoperable with a `void*` C pointer. All components are private. Thus a Fortran program may declare a variable that is interoperable with a C pointer by

```
Use, intrinsic :: iso_c_binding
Type( c_ptr ) :: foo_c_ptr
```

As this derived type is opaque, methods are provided to convert a Fortran variable to a C pointer and also a C pointer into a Fortran pointer. The former is

```
Use, intrinsic :: iso_c_binding
Type( c_ptr ) :: foo_c_ptr
foo_c_ptr = c_loc( foo )
```

The latter is

```
Use, intrinsic :: iso_c_binding
Type( c_ptr ) :: foo_c_ptr
Integer, Dimension( : ), Pointer :: foo_f_pointer
Call c_f_pointer( foo_c_ptr, foo_f_pointer, (/ 3, 2 /) )
```

which returns in `foo_f_pointer` a Fortran pointer to an integer array of shape (1:3,1:2) corresponding to the memory pointed at by the C pointer `foo_c_ptr`. Also provided is a variant on the associated intrinsic, `c_associated`, that deals with C rather than Fortran pointers.

This is a very simple mechanism. It is also extremely powerful and flexible, and it is the cornerstone of this work.

### 2.4 The Value Attribute

C passes arguments by value, which was not possible in Fortran prior to the 2003 standard. However the new standard allows a program to specify that an argument is passed by value by use of the `value` attribute for scalar dummy arguments. Note that this is not restricted to interoperating with C, and may be used within a purely Fortran program. To use this feature an `interface` must be in scope at the calling point. An example of its use may be found in the next section.

## 2.5 Invoking C Functions from Fortran

If a Fortran subprogram invokes a C function that function must be interoperable, just as for variables. To achieve this there must be an explicit interface in scope declared with the `bind` attribute; the interface

```
Interface
  Subroutine c_func( a, b, c ) Bind( C )
  Use, Intrinsic :: iso_c_binding
  Real    ( c_double ), Dimension( * ) :: a
  Integer( c_int    ), Value          :: b
  Type    ( c_ptr    ), Value          :: c
End Subroutine c_func
End Interface
```

enables interoperability with

```
void c_func( double *a, int b, void *c );
```

Note that the last two arguments are passed by value. The invocation is then simply

```
Real( c_double ), Dimension( 1:10 ) :: tigers
Integer( c_int )                    :: in
Type    ( c_ptr )                    :: africa
...
Call c_func( tigers, in, africa )
```

A similar mechanism may be used to invoke non-void C functions. In this case the interface should define a Fortran function rather than a subroutine.

## 3 System V IPC Facilities

In this section I shall introduce those features of the Posix compliant System V IPC facilities that are used by this work: shared memory segments and semaphores. Note that these facilities only work within an SMP node, and in this section, unless stated otherwise, references to synchronization and similar concepts should be understood to apply only within the SMP, that is intra-node. In this work all inter-node communication will be performed using MPI.

A fuller description of the facilities provided by System V IPC can be found in [6].

### 3.1 Shared Memory Segments

A shared memory segment is an area of memory that is accessible by one or more processes. It is created at run time, but this has little in common with the use of `malloc` and related functions in C. This is because more than one process may access this area of memory, resulting in a number of complications.

The first is that a shared memory segment is not automatically released by the OS when the program that created it terminates, whether that termination is successful or not. This is a result of the OS only knowing about the existence of processes, not

what the process is doing; if one process dies the OS does not know whether another process is accessing a given segment or not. Therefore it is **imperative** that after the last program that uses any user allocated shared memory segments terminates a check is made for any remaining segments, and if any are found they are deleted by use of the `ipcrm` command. If this is not done repeated execution will lead to more and more of the system memory being occupied by (unused) segments. Note this also implies another use for shared memory segments, chaining related programs together, however I shall not address this possibility further. Also note that on HPCx this deletion is performed automatically, but it should **not** be assumed that other machines will behave similarly.

The second problem is how to create the segments within the program. Two possible methods are:

1. all processes create the segment in unison;
2. one process creates the segment and then processes attach to it.

System V IPC uses the second choice; a segment is created with `shmget`, and processes then attach to it using `shmat`.

The prototype for `shmget` is

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget( key_t key, size_t size, int shmflg );
```

For `shmat` it is

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat( int shmid, const void *shmaddr, int shmflg );
```

In the above, one process uses `shmget` to create a shared memory segment of size `size` bytes, with permissions given by `shmflg`. An integer handle, here called `shmid`, is returned to identify the new segment. That process, and possibly other processes, then attaches to the segment by use of `shmat` which returns a pointer to the segment. The segment can then be accessed via this pointer.

However this leaves two questions unanswered:

1. How to choose which process should create the segment?
2. How should the creating process tell other processes the value of `shmid`?

In general 2 is a problem. However the focus of this work is MPI programs, for which the process rank provides an answer to 1, and `mpi_bcast` an answer to 2.

Segment destruction is the reverse of the above; all the processes detach from the segment, which is then freed by one of the processes. This achieved by `shmdt` and `shmctl` respectively.

### 3.2 Semaphores

System V IPC provides a very flexible mechanism by which such operations as locks and critical regions may be implemented. This mechanism is semaphores. For this work the only use of semaphores is the implementation of critical regions, as other synchronization mechanisms can be implemented by MPI.

Creation and destruction of a semaphore is a little simpler than that for shared memory segments as processes need not attach/detach. One process creates a semaphore by `semget`, and in this work the other processes in the SMP receive the integer handle that identifies the semaphore using `mpi_bcast`. Destruction is achieved by one process calling `semctl`.

Critical regions, a section of code which only one process within the SMP is allowed to execute at any given time, are implemented very simply with semaphores: They are used in an almost literal sense, as a flag. This implementation uses the `semop` system call. This routine is used for testing and setting/resetting the semaphore in an atomic manner. On entry to the critical region a process calls `semop`. Two possible behaviours are possible, dependent on the value of the semaphore:

1. If the semaphore is not set the process sets it and proceeds.
2. If it is set the process sleeps until the semaphore is reset, sets it and then proceeds. Note if multiple processes are sleeping only one is woken each time the semaphore is reset.

On exiting the critical region `semop` is called again to reset the semaphore. The use of `semop` to change the flag is necessary to ensure atomicity of the operations, and so avoid race conditions. The latter would occur if, for instance, the flag were held simply in a shared memory segment and multiple processes accessed it at once. While this may sound complex, for this work the complete implementation of critical regions is

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int fipc_c_crit_start( int sem )
{

    struct sembuf sem_op;

    sem_op.sem_num = 0;
    sem_op.sem_op  = -1;
    sem_op.sem_flg = 0;

    return semop( sem, &sem_op, 1 );

}
```

```
int fipc_c_crit_end( int sem )
{

    struct sembuf sem_op;

    sem_op.sem_num = 0;
    sem_op.sem_op  = 1;
    sem_op.sem_flg = 0;

    return semop( sem, &sem_op, 1 );

}
```

In fact as far as this implementation goes the biggest problem with semaphores is that the flexibility they provide is much too powerful for what is required!

Each instantiation of a critical region requires two system calls to `semop`, and each call may also send signals to a number of other processes within the SMP. Critical regions are, therefore, potentially very expensive. Further they most commonly occur within loops. Therefore they should be avoided wherever possible. I shall return to the expense of critical regions later.

## 4 Bringing It All Together: FIPC

While sections 2 and 3 introduced the basic tools, here I shall very briefly describe the implementation and use of a general purpose Fortran module that is intended to enable the easy, portable and efficient use of shared memory segments in MPI codes. This module is called FIPC.

The design of the API to FIPC owes a lot to MPI. After all if an API is as successful as MPI there must be something going for it! Indeed in some ways FIPC can be viewed as an extension of MPI; it extends the concept of the MPI communicator to include shared memory concepts. This extended communicator I shall term a **context**.

### 4.1 Contexts

It is best to be clear at the outset what a context is. Ultimately it is built on an MPI communicator. Thus a context spans a number of processes. However for the shared memory aspects it must also contain information about the underlying shared memory architecture, i.e. how the processes map onto the SMP nodes. Therefore a context also spans a number of SMP nodes. However it is important to note that because a context contains one process in an MPI job on a given SMP node it does **not** necessarily mean that it contains all the processes in the job on that node. That will depend upon which processes the underlying MPI communicator spans.

The vast majority of FIPC operations occur within a context, in a similar manner to MPI and communicators. When an operation occurs within a context it should be

understood that all SMP nodes spanned by the context behave in unison. For example creation of a shared memory segment within a context implies that all SMP nodes spanned by the context will create a shared memory segment. As another example a reduction operation within a context involves a reduction across shared memory segments on each of the nodes spanned by the context. In this sense virtually all FIPC operations are collective.

## 4.2 Use of FIPC

FIPC allows the creation of as many shared memory segments as the underlying OS allows. Their size is limited by the amount of available memory and the maximum allowed by the underlying OS. Manipulation of the data within those segments is extremely simple, as will be shown below. Collective operations that operate across shared memory segments held on different SMP nodes are provided, as are a number of utility operations. Critical regions are provided for synchronization within an SMP node. Virtually all operations occur within an FIPC context, similar to MPI and communicators. At present the following data types are supported:

- `Integer( c_int )` scalars, i.e. default Integers.
- One dimensional arrays of `Integer( c_int )`
- Two dimensional arrays of `Integer( c_int )`
- `Real( c_double )` scalars, i.e. “Double Precision” scalars
- One dimensional arrays of `Real( c_double )`
- Two dimensional arrays of `Real( c_double )`

Extension to other Fortran intrinsic data types is very straightforward. The software may be obtained from the author.

FIPC is accessed by `Use` of the module in the appropriate subprograms. It is initialized by calling `fipc_init`, and finalized by `fipc_finalize`. After initialization the context `fipc_ctxt_world` is available. This spans all SMP nodes in the MPI job, and contains all the processes in the job. It is therefore analogous to `mpi_comm_world`. Note that unlike MPI this is not an integer handle but an opaque structure of type `fipc_ctxt`. It is possible to create user defined contexts by `fipc_ctxt_create`, and destroy them with `fipc_ctxt_destroy`. Shared memory segments are created by `fipc_seg_create`. For example

```
Real( c_double ), Dimension( :, : ), Pointer :: a
Call fipc_seg_create( (/ 3, 2 /), fipc_ctxt_world, a, error )
```

creates a shared memory segment of shape (1:3,1:2) which is pointed at by `a`. As described above one segment is created on **each** of the SMP nodes spanned by `fipc_ctxt_world`. All the nodes and all the processes in `fipc_ctxt_world`, i.e. all the processes in the job, have access to the segment on the SMP in which the process resides.

The variable `a` is a Fortran pointer. Thus to access the (2,1) element all that is required is

```
b = a( 2, 1 )
```

and to update an element

```
a( 3, 2 ) = a( 3, 2 ) + 5.0_c_double
```

Indeed, as `a` is a primary Fortran type all the power of the language is available for its manipulation! The resulting use of shared memory segments could hardly be easier, or more efficient. As an aside this could be used for extremely low latency communication within a node.

Of course the programmer must be aware that `a` points at memory that may be modified by other processes<sup>ii</sup>. For instance, in the example directly above, a race condition may occur if another process is writing to `a` at the same time. For this reason critical regions are provided. Writes to `a` may be protected via

```
Call fipc_critical_start( fipc_ctxt_world, error )
a( j, i ) = a( j, i ) + value
Call fipc_critical_end( fipc_ctxt_world, error )
```

Note that critical regions also require a context. This allows for synchronization across less than the full SMP node. It also allows for different contexts to protect different parts of a data structure, which could potentially increase performance by decreasing the amount of synchronization with an SMP node.

One other synchronization routine is provided, `fipc_barrier`. This synchronizes all processes that are in the context within each SMP node spanned by the context, which is useful to ensure all writes to a shared memory segment have completed. Note that it only blocks within an SMP node, and so should be a lot cheaper than a full synchronization across the whole MPI job.

Other routines include

- `fipc_allreduce` which performs reduction operations across shared memory segments within the given context, and is analogous to `mpi_allreduce`
- `fipc_ctxt_size` which returns the number of SMP nodes spanned by a context. This is **not** a collective operation.
- `fipc_ctxt_rank` which returns the rank of the calling SMP node. This is **not** a collective operation.
- `fipc_ctxt_base_comm` which returns the communicator from which the context was created. This is **not** a collective operation. Finally
- `fipc_ctxt_node_comm` which returns a communicator that contains the processes on the calling SMP node that are members of this context. This is **not** a collective operation.

---

<sup>ii</sup> Fortran 2003 also provides the `volatile` attribute which may be of use, and is also now widely implemented

### 4.3 Implementation

Though the actual details of the implementation need not be an issue, as a definition of the API is all that the user need know, it is worth emphasizing how the FIPC module builds on standardized elements to provide a portable, powerful and usable solution. The standardized System V IPC component provides access to the shared memory facilities of the OS, the standardized C interoperability component provides a portable way for Fortran application programmers to access these facilities, and MPI provides the standard way to communicate both between nodes and, where necessary, within them. To highlight just one feature, the availability of `c_f_pointer` is crucial to FIPC. Without it the Fortran programmer would not have a simple method to access the memory in the shared segment, but with it such accesses become both extremely simple and very efficient.

## 5 FIPC in Action

In this section I shall review some results from the use of FIPC for Hartree-Fock calculations in GAMESS-UK, a standard quantum chemistry code. The standard parallel implementation of such codes requires very large amounts of replicated memory. They are therefore ideal candidates for using shared memory segments, and so FIPC.

### 5.1 GAMESS-UK

GAMESS-UK [7] represents a typical established electronic structure code, comprising some 800K lines of Fortran that permits a wide range of computational methodology to be applied to molecular systems. For Hartree-Fock calculations only two sections of the code take a significant amount of time. One is almost purely dense linear algebra. This can be addressed very effectively by a number of methods, for example the use of ScaLAPACK [8] or Global Arrays [9]. The second involves construction of one of the matrices used in the linear algebra, the Fock matrix  $\mathbf{F}$ . It is this step that uses the replicated memory.

The calculation of  $\mathbf{F}$ , the Fock build, requires the evaluation of a very large number of integrals of the form

$$\langle ij||kl \rangle = \int dr_1 \int dr_2 \Phi_i(\mathbf{r}_1) \Phi_j(\mathbf{r}_1) |\mathbf{r}_1 - \mathbf{r}_2|^{-1} \Phi_k(\mathbf{r}_2) \Phi_l(\mathbf{r}_2)$$

where  $\Phi_i(\mathbf{r})$  is a contracted Gaussian type orbital (GTO)

$$\Phi_i(\mathbf{r}) = \sum_j c_{ij} (x-x_i)^{\lambda_i} (y-y_i)^{\mu_i} (z-z_i)^{\nu_i} e^{-\alpha_j |r-r_i|^2}$$

The set of  $\Phi_i(\mathbf{r})$  is termed the basis set, and for a given molecule the number of GTOs in the basis set determines both the quality and the cost of the calculation.

An element of the Fock matrix is given by

$$F_{ij} = F_{0,ij} + \sum_{kl} P_{kl} \langle ij||kl \rangle - \frac{1}{2} \sum_{kl} P_{kl} \langle ik||jl \rangle$$

where  $\mathbf{F}_0$  and  $\mathbf{P}$ , the core Hamiltonian and the Density matrix, are constant throughout the Fock build. As the whole method of solution, Self-Consistent Fields (SCF), is iterative the Fock build may have to be performed many times.

The key to efficient evaluation of the Fock matrix is not the efficient evaluation of each individual integral, but rather the efficient avoidance of calculating an integral. Three methods are used. First  $\mathbf{F}$  is symmetric, so only one half needs to be calculated. Second it is clear from the form of the integral that there are a number of permutation symmetries

$$\langle ij||kl \rangle = \langle ji||kl \rangle = \langle kl||ij \rangle = \dots$$

Thus each integral contributes to up to 8 different Fock matrix elements, and needs to be calculated only once. Finally an upper bound can be put on each integral by using the Schwarz inequality

$$|\langle ij||kl \rangle| \leq [|\langle ij||ij \rangle \langle kl||kl \rangle|]^{1/2}$$

This can be used to reject very small integrals if the set of  $\langle kl||kl \rangle = T_{kl}$  is pre-calculated.

Combining the above, for a basis set of size  $N$  and  $T_{\max} = \text{Maxval}(T_{kl})$  the standard parallel algorithm for a Fock build is:

```

F = 0.0
Call get_next( i_next, j_next )
Do i = 1, n
  Do j = 1, i
    If( i /= i_next .Or. j /= j_next ) Then
      Cycle
    End If
    If( Abs( T_ij * T_max ) < tol ) Then
      Cycle
    End If
    Do k = 1, i
      Do l = 1, Min( k, j )
        If( Abs( T_ij * T_kl ) < tol ) Then
          Cycle
        End If
        I_ijkl = <ij||kl>
        F_ij = F_ij + P_ij * I_ijkl
        F_kl = F_kl + P_kl * I_ijkl ! Uses permutation symmetry
        F_ik = F_ik - 0.5 * P_jk * I_ijkl
        F_il = F_il - 0.5 * P_jl * I_ijkl ! Permutation symmetry
        F_jk = F_jk - 0.5 * P_ik * I_ijkl ! Permutation symmetry
        F_jl = F_jl - 0.5 * P_il * I_ijkl ! Permutation symmetry
      End Do
    End Do
  End Do
End Do

```

```
        End Do
        Call get_next( i_next, j_next )
    End Do
End Do
Call global_sum( F )
F = F + F0
```

In the above `get_next` implements a load balancing scheme, which is often dynamic.

The algorithm can be seen to scale as  $O(N^2)$  to  $O(N^4)$  depending on how many of the integrals are rejected. It can also be seen that the access patterns for **F**, **P** and **T** are far from simple, and for this reason it is these matrices that are usually replicated. The memory for these matrices scales as  $O(N^2)$ , and so the amount of replicated data required can be very large. Therefore it is these matrices that are the target for FIPC. For technical reasons not covered here in practice **T** is somewhat smaller than **F** or **P**. However it is still large enough to cause concern.

Adaptation of the above algorithm to use FIPC is fairly straightforward. **P** and **T** are only ever read, and so once generated in the shared memory segment no further special precautions are required. However the segment holding **F** will be written to by a number of processors. Therefore a critical region must be used for each write to **F** in order to avoid race conditions. This is deep within the inner loop, which raises concerns about the efficiency of the procedure. To help alleviate this the implementation uses a fixed size buffer which is filled with a large number of integrals corresponding to different values of *i*, *j*, *k* and *l*, and only once the buffer is filled is the Fock matrix updated. This reduces the number of critical regions, and hence system calls and synchronizations, that are required.

## 5.2 Results

Three molecular systems were used to test the FIPC implementation. The first was nitrobenzene,  $C_6H_5NO_2$  (PhNO<sub>2</sub>). This is a very small system which was used both to validate the method and to investigate the performance concerns for small systems. The second is an artificial system comprising 1000 Hydrogen atoms on a primitive cubic lattice. This was used to investigate the performance for larger systems. Finally a number of systems derived from the enzyme Isocitrate Lyase were studied. These are very large cases, and were chosen to show what the FIPC based method is capable of. In each case a number of different basis sets was used. All results were generated on the Phase 2 HPCx system, which was a cluster of IBM p690+ 32-way SMP nodes, each node having 32 GByte of memory.

### 5.2.1 Nitrobenzene ( $\text{PhNO}_2$ )

Figure 1 shows the performance of the FIPC Fock build for three different basis sets. The performance is measured relative to the time taken for the standard Fock build; a value of 1 indicates that the FIPC build is taking the same time as the standard method, while 2 means that the FIPC build is taking twice as long as the standard. The runs were performed on 32 processors, i.e. 1 node. Three different basis sets were used: 6-31G (86 basis functions), 6-31G\* (130) and tzvp (201). The performance is shown as a function of the “cluster size,” which is the number of processors that share a segment; using the context method it is possible to use FIPC to set up 16 segments each of which is shared by two processors, or 8 segments each shared by 4, etc., and at the end of the calculation the segments are summed. Effectively one may think of the cluster size as representing a logical SMP constructed by the software within the real SMP node. It therefore has some similarities to the LPARs on the Phase 1 machine.

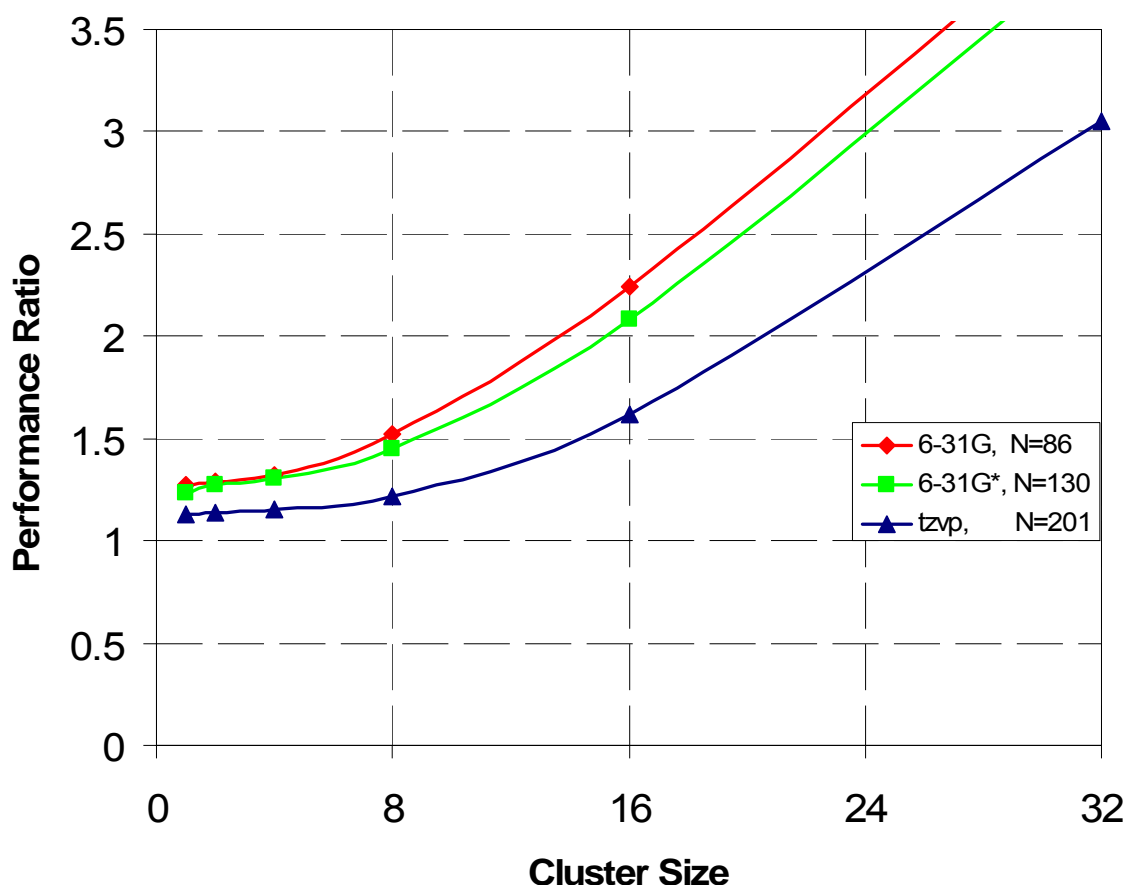


Figure 1 - The Performance ratio of the FIPC based method compared to the standard parallel code for  $\text{PhNO}_2$  as a function of cluster size and basis set

For this very small system the overhead due to the critical regions can be seen very clearly, especially for the larger cluster sizes. However increasing the size of the basis, and hence the cost of the calculation, noticeably reduces this overhead. For the tzvp basis the FIPC build is only slightly slower than the standard method for cluster sizes up to about 8. This was very encouraging as nitrobenzene really is a very small system.

That the overhead should reduce as the size of basis set is increased is a little surprising. The time between the synchronization caused by the critical region should be fixed as the size of the buffer used to store the integrals is fixed. What was found in practice for nitrobenzene was that buffer was never filled, even for the tzvp basis. As a result the larger basis did imply a longer time between critical regions, and hence a proportionately lower overhead.

Figure 2 again shows the performance of the FIPC Fock build relative to the standard parallel method, but this time the basis set is held constant at the 6-31G level and the number of processors is varied.

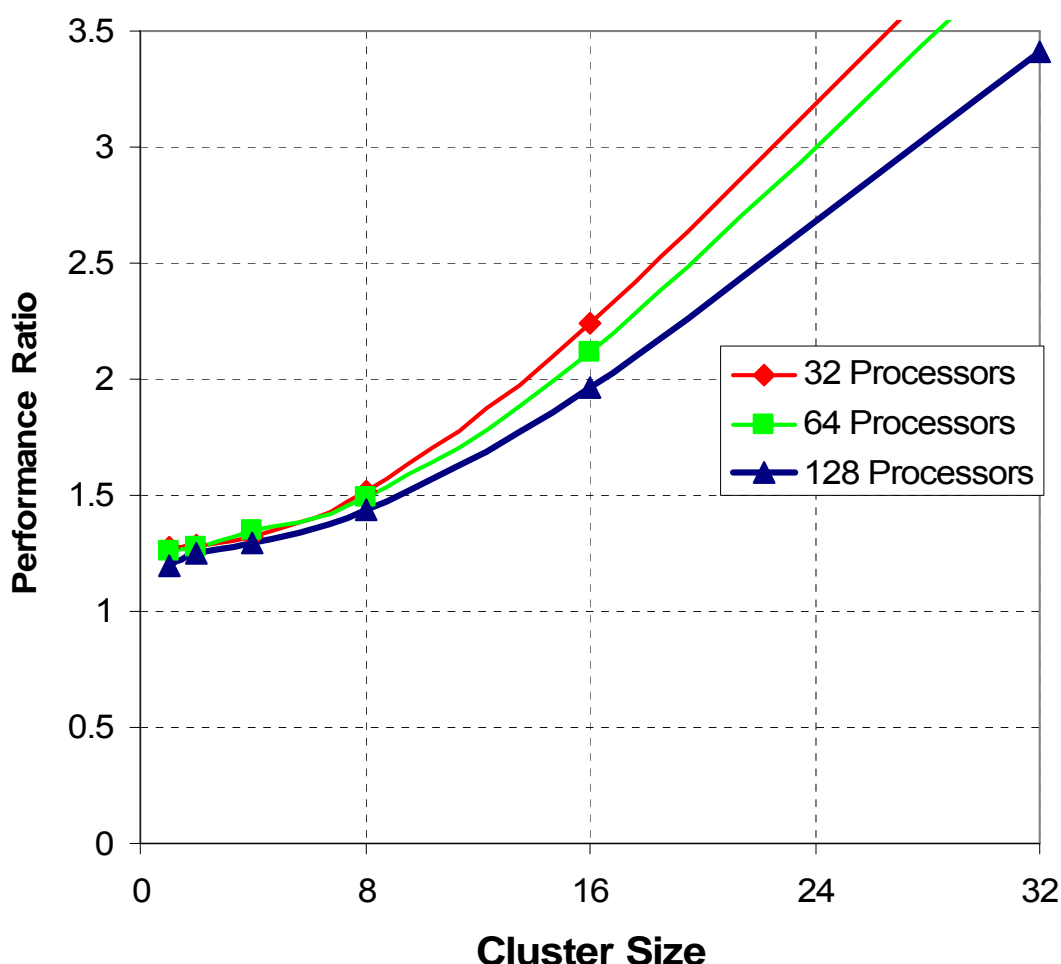


Figure 2 - The Performance ratio of the FIPC based method compared to the standard parallel code for  $\text{PhNO}_2$  as a function of the number of processors

It can be seen that there is a small improvement with increasing number of processors, especially for large cluster size. This is surprising. FIPC only affects the behaviour within an SMP node, so the expected result would be that the overhead due to FIPC would be independent of the number of SMP nodes. The effect is, however, very small, and is not seen for large systems, which are the cases of interest.

### 5.2.2 Primitive Cubic Hydrogen (PCH)

To investigate larger cases the artificial system of 1000 hydrogen atoms on a primitive cubic lattice was constructed. Though artificial this system has the very nice property of regularity, so ensuring that any peculiarities of behaviour are purely due to the computational method and not the chemical make up of the system. Three basis sets were used; STO-3G (1000 basis functions), 3-21G (2000) and 3-21G\*\* (5000). The results are shown in figure 3. The runs were performed on 128 processors, and again the performance is measured relative to the standard method. It can be seen that for this much larger system the performance graphs are essentially flat – note the scale on the performance ratio axis. Yet for the two larger basis sets the FIPC build is slightly *faster* than the standard method, and what is more the best performance is seen for cluster sizes larger than 1. Both effects are small, but for both the data are very clear.

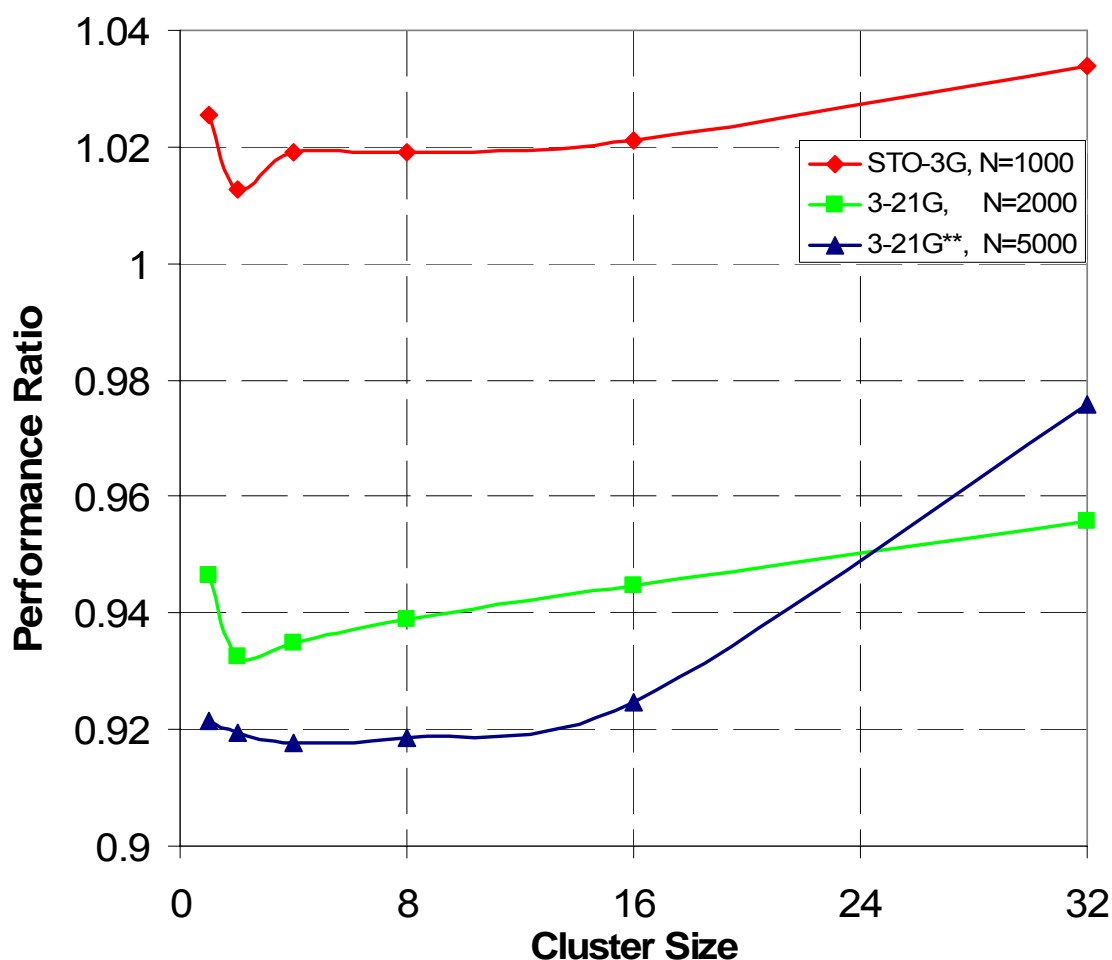


Figure 3 -The performance for PCH as a function of cluster size and basis set

One probable cause for both effects is that if **F**, **P** and **T** are held in shared memory segments not only are there fewer replicated copies per node, thus reducing memory usage, there are also fewer copies in the various shared levels of cache, so effectively *increasing* the size of the cache. However this does not explain why FIPC is also faster for a cluster size of 1, and it is not obvious why this is observed.

On the other hand the more rapid degradation in performance for 3-21G\*\* when compared to 3-21G is readily explained. The smaller basis contains only s-type GTOs, which have 1 component. Therefore each entry in the integral buffer is precisely  $1*1*1*1=1$  element. The larger basis also includes p functions, each of which have x, y and z components. Therefore each entry in the integral buffer can contain up to  $3*3*3*3=81$  elements. The buffer therefore fills up more quickly, leading to more frequent synchronizations, so slowing down the calculation at large cluster sizes. On the other hand the integrals involving p functions are more complicated to calculate than those only including s functions, and so the former benefit more from the effective increase in cache size due to data reuse. Thus at small cluster size 3-21G\*\* is accelerated more than 3-21G.

For the 3-21G basis set the best performance is at a cluster size of 2, and for the 3-21G\*\* it is at a cluster size of 4. However the curves are very flat, and for this system any cluster size will not lead to a marked degradation in performance.

The pragmatic conclusion is that the overhead due to critical regions for systems larger than around 1000 basis functions is, at worst, minimal for all cluster sizes on this system.

### 5.2.3 Isocitrate Lyase

Isocitrate lyase is a key enzyme in persistent Mycobacterium Tuberculosis, and is being studied by Joop van Lenthe and co-workers [10] as part of a project to study potential inhibitors for chronic tuberculosis. It is a huge system. Full quantum calculations on the whole molecule are simply not possible now, and will not be for many years hence. However, like most proteins, it has one active site where the chemistry of interest occurs. Therefore a hierarchy of systems was defined by including all atoms in the enzyme that are within a given radius of the magnesium ion in the active site. Even this simplification rapidly leads to very large systems. Table 1 shows the number of basis functions required for each system and the approximate amount of memory required to store the important replicated matrices.

Radius/Å	Number of Basis Functions	Memory/Gbyte
12	4210	0.28
14	5901	0.56
16	7427	0.88
18	8980	1.29
20	11742	2.21

Table 1 - Size of and memory requirements for the Isocitrate Lyase calculations

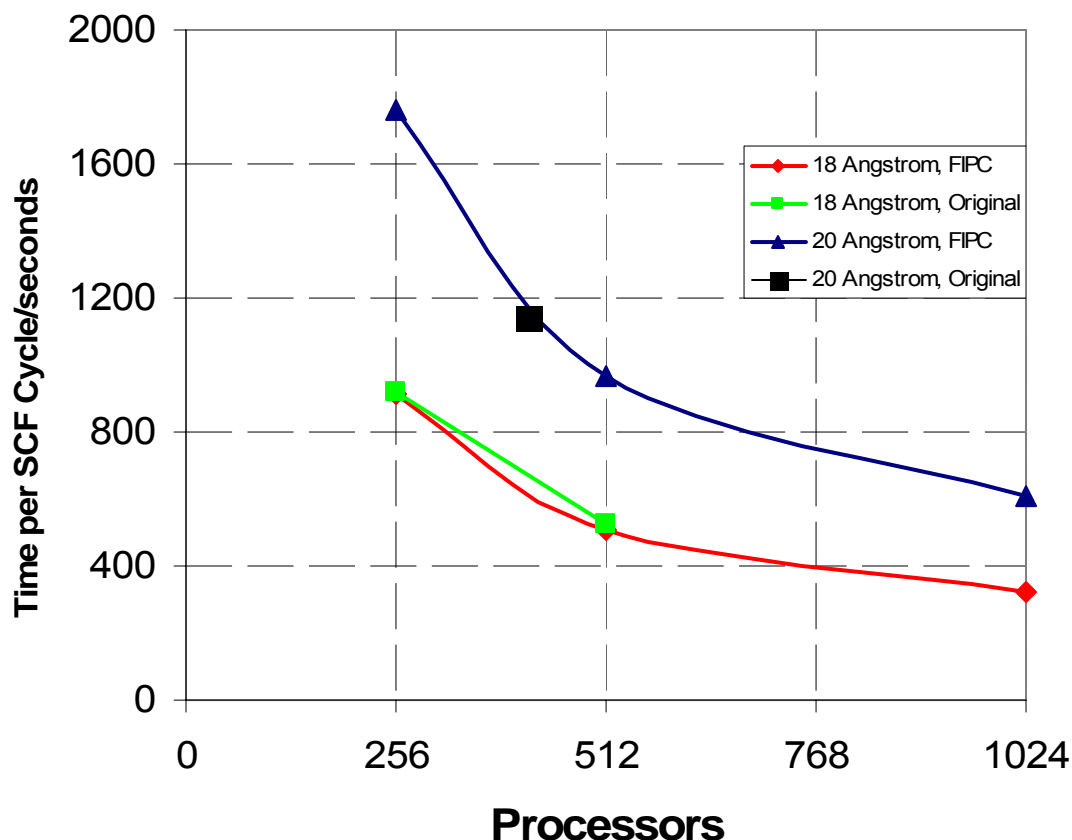


Figure 4 - The time per SCF Cycle for Isocitrate Lyase

Figure 4 shows the time per SCF iteration for the 18 Å and 20Å cases. In all cases a cluster size of 16 was used.

Where comparison is possible it can be seen that the performance of the FIPC Fock builder and the original are virtually identical. However the maximum number of processors upon which the original can be run is about 512 for the 18Å calculation and only 416 for the 20Å case. The reason why is clear from table 1. For the original method the limit on the size of system that can be studied is the memory available to each process. The 18Å and 20Å calculations require more than 1 GByte per process. Therefore on the Phase 2 machine, on which each 32-way SMP node had 32 Gbytes of memory, the *only* way of obtaining more than 1 GByte per process was to underpopulate the nodes, i.e. to use less than all the 32 processors. The FIPC builder, however, is only limited by the available memory **per node**. Therefore it can use all 32 processors in the node, and can run on 1024 processors in total.

FIPC, therefore, enables a shorter time to solution. It is also much cheaper. The charging model for HPCx is, and has always been, per SMP node. Applying this scheme figure 5 shows the costs of the calculations relative to the charge for running the FIPC calculation on 256 processors.

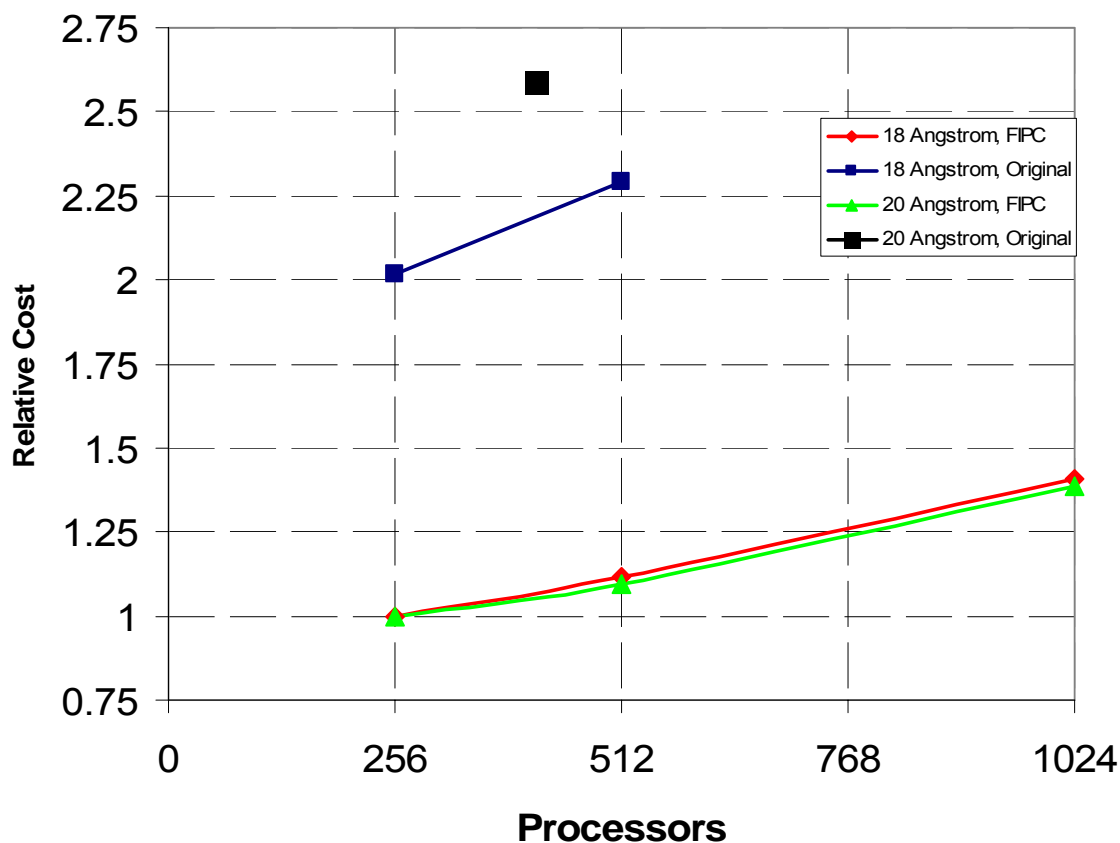


Figure 5 - The relative cost of the Isocitrate Lyase Calculations

It can be seen that for a given number of processors the FIPC calculation is half the cost of the original or better, as it uses the resources available much more effectively.

The FIPC Fock Builder has been pushed further. On 512 processors of the Phase 2a machine it took 3764.5 seconds to perform an SCF cycle on an 18Å radius Isocitrate Lyase system using a 6-31G\*\* basis set. This calculation uses 22440 basis functions.

## 6 Conclusions

It has been shown that System V IPC facilities can provide a very effective solution to the replicated memory problem found in many quantum chemistry applications. It has also been shown that use of features from the new Fortran standard, namely interoperability with C, allow very easy access to IPC facilities for the Fortran application programmer. The resulting code has been shown to be capable of dramatically out performing the original as it makes better use of the resources available on HPCx.

## 7 Acknowledgements

I would like to thank Stephen Booth of EPCC for his help in becoming acquainted with the System V IPC routines and their use.

For the work with GAMESS-UK I would like to thank Joop van Lenthe of the University of Utrecht, and members of the Quantum Chemistry group at CCLRC Daresbury Laboratory, in particular Huub van Dam, Paul Sherwood and Martyn Guest. They were involved in many useful discussions before, during and after this work, and they also provided the Isocitrate Lyase test cases above.

## References

- [1] ISO/IEC 1539-1:2004
- [2] OpenGL - The Industry Standard for High Performance Graphics, <http://www.opengl.org/>
- [3] The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition <http://www.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html>
- [4] Simple DirectMedia Layer <http://www.libsdl.org/>
- [5] *Fortran 95/2003 Explained*, M.Metcalf, J.Reid, M.Cohen, Oxford University Press, ISBN 0-19-852693-8
- [6] *Beginning Linux Programming*, N.Matthew and R.Stones, Wrox, ISBN 0-7645-4497-7
- [7] GAMESS-UK is a package of *ab initio* programs written by M.F. Guest, J.H. van Lenthe, J. Kendrick, K. Schoeffel and P. Sherwood, with contributions from R.D. Amos, R.J. Buenker, M. Dupuis, N.C. Handy, I.H. Hillier, P.J. Knowles, V. Bonacic-Koutecky, W. von Niessen, R.J. Harrison, A.P. Rendell, V.R. Saunders, and A.J. Stone. The package is derived from the original GAMESS code due to M. Dupuis, D. Spangler and J. Wendoloski, NRCC Software Catalog, Vol. 1, Program No. QG01 (GAMESS), 1980.
- [8] The ScaLAPACK Project <http://www.netlib.org/scalapack>
- [9] The Global Arrays toolkit <http://www.emsl.pnl.gov/docs/global/>
- [10] *Large Scale Computing with GAMESS-UK*, J.H. van Lenthe, M.F. Guest, H.J.J. van Dam, I.J. Bush, Abstr Paper Am Chem Soc Natl Meet 230 U1309 (2005)