(25 March 2000)

Section 5 - Programmer's Reference

This section describes features of the GAMESS implementation which are true for all machines. See the section 'hardware specifics' for information on each machine type. The contents of this section are:

Installation overview

GAMESS will run on a number of different machines under FORTRAN 77 compilers. However, even given the F77 standard there are still a number of differences between various machines. For example some machines have 32 bit word lengths, requiring the use of double precision, while others have 64 bit words and are used in single precision.

Although there are many types of computers, there is only one (1) version of GAMESS.

This portability is made possible mainly by keeping machine dependencies to a minimum (that is, writing in F77, not vendor specific language extensions). The unavoidable few statements which do depend on the hardware are commented out, for example, with "*IBM" in columns 1-4. Before compiling GAMESS on an IBM machine, these four columns must be replaced by 4 blanks. The process of turning on a particular machine's specialized code is dubbed "activation".

A semi-portable FORTRAN 77 program to activate the desired machine dependent lines is supplied with the GAMESS package as program ACTVTE. Before compiling ACTVTE on your machine, use your text editor to activate the very few machine dependent lines in ACTVTE before compiling it. Be careful not to change the DATA initialization!

The task of building an executable form of GAMESS is this:

activate		compile		link	
*.SRC	--->	*.FOR	--->	*.OBJ	---> *.EXE
source code		FORTRAN code		object code	executable image

where the intermediate files *.FOR and *.OBJ are discarded once the executable has been linked. It may seem odd at first to delete FORTRAN code, but this can always be reconstructed from the master source code using ACTVTE.

The advantage of maintaining only one master version is obvious. Whenever any improvements are made, they are automatically in place for all the currently supported machines. There is no need to make the same changes in a plethora of other versions.

The control language needed to activate, compile, and link GAMESS on your brand of computer is probably present on the distribution tape. These files should not be used without some examination and thought on your part, but should give you a starting point.

There may be some control language procedures for one computer that cannot be duplicated on another. However, some general comments apply: Files named COMP will compile a single module. COMPALL will compile all modules. LKED will link together an executable image. RUNGMS will run a GAMESS job, and RUNALL will run all the example jobs.

The first step in installing GAMESS should be to print the manual. If you are reading this, you've got that done! The second step would be to get the source code activator compiled and linked (note that the activator must be activated manually before it is compiled). Third, you should now compile all the source modules (if you have an IBM, you should also assemble the two provided files). Fourth, link the program. Finally, run all the short tests, and very carefully compare the key results shown in the 'sample input' section against your outputs. These "correct" results are from a IBM RS/6000, so there may be very tiny (last digit) precision differences for other machines. That's it!

Before starting the installation, you should read the pages decribing your computer in the 'Hardware Specifics' section of the manual. There may be special instructions for your machine.

Files for GAMESS

*.DOC	These are the TEXT versions of the manual you are reading now. Since you have presumably already printed out the PostScript version there is no need to print these out, but they are useful for searching online with a text editor like VI or EMACS...
*.SRC	source code for each module
*.ASM	IBM mainframe assembler source
*.C	C code used by some UNIX systems.
EXAM*.INP	29 short test jobs (see TESTS.DOC).

These are files related to some utility programs:

ACTVTE.CODE	Source code activator. Note that you must use a text editor to MANUALLY activate this program before using it.
MBLDR.*	model builder (internal to Cartesian)
CARTIC.*	Cartesian to internal coordinates
CLENMO.*	cleans up $VEC groups

There are files related to X windows graphics. See the file INTRO.MAN for their names.

The remaining files are command language for the various machines.

*.COM	VAX command language. PROBE is especially useful for persons learning GAMESS.
*.MVS	IBM command language for MVS (dreaded JCL).
*.CMS	IBM command language for CMS. These should be copied to filetype EXEC.
*.CSH	UNIX C shell command language. These should have the "extension" omitted, and have their mode changed to executable.

Running Distributed Data Parallel GAMESS

It is difficult to write a description of the parallel nature of GAMESS that separates what is important for the installer of GAMESS or programmers of GAMESS from the end user. Users of GAMESS should read most of this section, skipping only the most technical parts, in order to be able to effectively run this program.

Efficient use of GAMESS requires an understanding of three critical issues: The first is the difference between two types of memory (replicated MEMORY and distributed MEMDDI) and how these relate to the physical memory of the computer which you are using. Second, you must understand to some extent the degree to which each type of computation scales so that the proper number of nodes is selected. Finally, most systems run two copies of GAMESS on each processor, and if you read on you will find out why this is so.

Since all code needed to implement the Distributed Data Interface (DDI) is provided with the GAMESS source code distribution, the program compiles and links ready for parallel execution on all machine types. Of course, you may choose to run on only one processor, in which case GAMESS will behave as if it is a sequential code, and the full functionality of the program is available.

Below you will find the following topics:

- parallelization history
- DDI process/memory schematic
- memory allocations and check jobs
- transport protocols and installation
- representative performance examples
- a very few programming details

* * *

We began to parallelize GAMESS in 1991 as part of the joint ARPA/Air Force piece of the Touchstone Delta project. Today, nearly all ab initio methods run in parallel, although some of these still have a step or two running sequentially only. Only the MP2 energy for MCSCF, and RHF CI gradients have no parallel method coded. We have not parallelized the semi-empirical MOPAC runs, and probably never will. Additional parallel work is in progress under a DoD CHSSI software initiative which "kicked off" in 1996. This has already led to the DDI-based parallel MP2 gradient program, after development of the DDI programming toolkit itself.

In 1991, the parallel machine of choice was the Intel Hypercube although small clusters of workstations could also be used as a parallel computer. In order to have the best blend of portability and functionality, we chose in 1991 to use the TCGMSG message passing library rather than one of the early vendor's specialized libraries. As the major companies began to market parallel machines, and as MPI version 1 emerged as a standard, we began to use MPI on some equipment in 1996, while still using the very resilient TCGMSG library on everything else. However, in June 1999, we retired our old friend TCGMSG when the message passing library used by GAMESS changed to the Distributed Data Interface, or DDI. We will discuss later the low level message transports which DDI relies on: SHMEM, TCP/IP sockets, or MPI-1. Two people have been extremely influential upon the current parallel methodology. Theresa Windus, a graduate student in the early 1990s, created the first parallel versions. Graham Fletcher, a postdoc in the late 1990s, is responsible for the addition of distributed data programming concepts.

* * *

DDI contains the usual parallel programming calls, such as initialization/closure, point to point messages, and the collective operations global sum and broadcast. These simple parts of DDI support all parallel methods developed in GAMESS from 1991-1999, which were based on replicated storage rather than distributed data. However, DDI also contains additional routines to support distributed memory usage.

DDI attempts to exploit the entire machine in a scalable way. While our early work concentrated on exploiting the use of p processors and p disks, it required that all data in memory be replicated on every one of the p nodes. The use of memory also becomes scalable only if the data is distributed across the aggregate memory of the parallel machine. The concept of distributed memory is contained in the Remote Memory Access portion of MPI version 2, but so far MPI-2 is not available from American computer vendors. The original concept of distributed memory was implemented in the Global Array toolkit of Pacific Northwest National Laboratory (see http://www.emsl.pnl.gov/pub/docs/global).

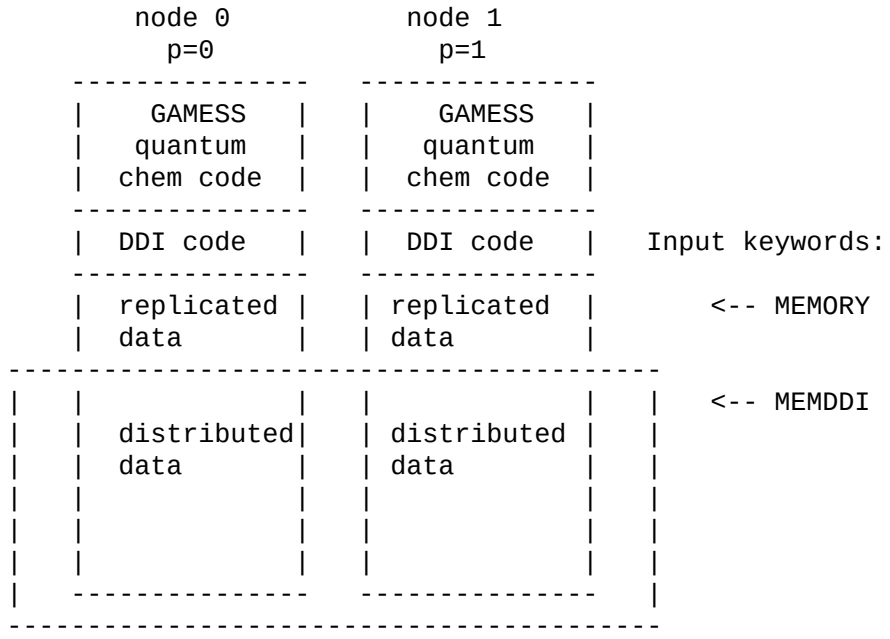
Basically, the idea is to provide three subroutine calls to access memory on remote nodes: PUT, GET, and ACCUMULATE. These give access to a class of memory which is assumed to be slower than local memory, but faster than disk:

```
    <--- fastest                        slowest --->
registers cache(s) local_memory remote_memory disks tapes
    <--- smallest                       biggest --->
```

Because DDI accesses memory on other nodes by means of an explicit subroutine call, the programmer is aware that a message must be transmitted. This awareness of the access overhead should encourage algorithms that transfer many data items in a single message. Use of a subroutine call to reach remote memory is a recognition of the non-uniform memory access (NUMA) nature of parallel computers. In other words, the Distributed Data Interface (DDI) is an explicitly message passing implementation of global shared memory.

In order to have one node pass data items to a second node when the second node needs them, without any significant delay, the computing job on the first node must interrupt its computation briefly to furnish the data. This type of communication is referred to as "one sided messages" or "active messages" since the first node is an unwitting participant in the process, which is driven entirely by the requirements of the second node.

The Cray T3E has a library named SHMEM to support this type of one sided messages (and good hardware support for this too) so, on the T3E, GAMESS runs as a single process per CPU. Its memory image looks like this:



where the box drawn around the distributed data is meant to imply that a large data array is residing in the memory of all nodes, in this example, half on one and half on the other. At the present time, the DDI routines support only two dimensional FORTRAN arrays, organized so that columns are kept on a single node's memory. Up to 10 matrices may be distributed in this fashion.

Note that the input keyword MEMORY gives the amount of storage used to duplicate small matrices on every node, while MEMDDI gives the -total- distributed memory required by the job. Thus, if you are running on p nodes, the memory that is used on any given node is

total on any 1 node = MEMORY + MEMDDI/p

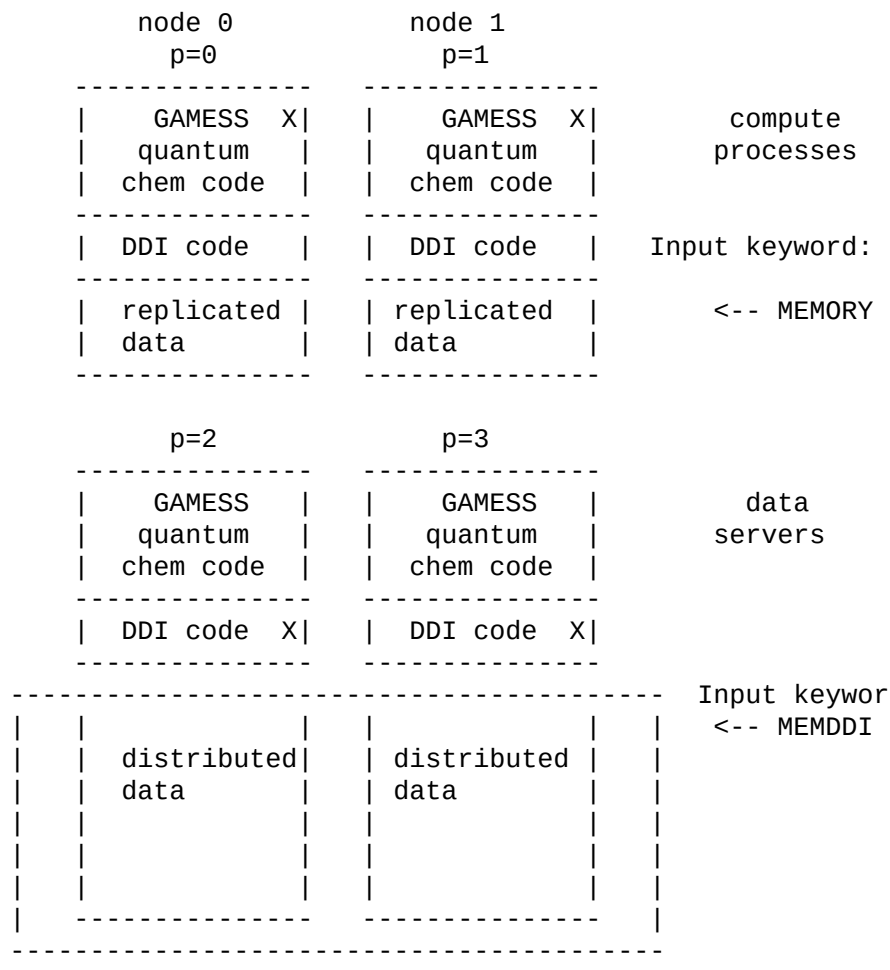
Since MEMDDI is very large, its units are in millions of words. The keyword MEMORY is in units of words (64 bit quantity) and so you must either convert units carefully or use the MWORDS synonym for MEMORY (for which the units are also millions of words). Since good execution speed requires that you not exceed the physical memory belonging to your nodes, it is important to understand that when MEMDDI is large, you will need to choose a sufficiently large number of nodes to keep the memory on any 1 node reasonable.

To repeat, the DDI philosophy is to add more processors not just for their compute performance or extra disk space, but also to aggregate a very large total memory. Bigger problems will require more nodes to obtain sufficiently large total memories! We will give an example of how you can estimate the number of nodes a little ways below.

If the GAMESS task running as process p=1 in the above example needs some values previously computed, it issues a call to DDI_GET. The DDI routines in process p=1 then figure out where this "patch" of data in the big rectangular distributed storage actually resides. Suppose this is on process p=0. The DDI routines in p=1 send a message to p=0 to interupt its computations, after which p=0 sends a bulk data message to process p=1's buffer. This buffer resides in part of the replicated storage of p=1, where computations can occur. Thus distributed data is accessed only by DDI_GET, DDI_PUT (its counterpart for storage of data items), and DDI_ACC (which accumulates new terms into the distributed data). Note that the

quantum chemistry layer of process p=1 was sheltered from most of the details regarding which node owned the patch of data that process p=1 wanted to obtain. These details are managed by the DDI layer. It is the programmer's responsibility to minimize the number of GET/PUT/ACC calls, and to design algorithms that maximize the chance that the patches of data are actually within the local node's portion of the distributed data. Note that with the exception of DDI_ACC's simple addition, no arithmetic is done directly upon the distributed data. Instead, DDI_GET and DDI_PUT should be thought of as analogous to the FORTRAN READ and WRITE statements that transfer data between disk storage and local memory where computations may occur.

Since MPI-2 is unavailable, and vendor specific "one sided messaging" libraries such as the T3E's SHMEM are scarce, all other platforms adopt the following strategy. It involves two GAMESS processes running on every node:



The first half of the processes do quantum chemistry, and the X indicates that they spend most of their time executing some sort of chemistry. Hence the name "compute process". Soon after execution, the second half of the processes call a servicing DDI routine which consists of an infinite loop to deal with GET, PUT, and ACC requests until such time as the job ends. The X shows that these "data servers" execute only DDI support code. This makes the data server's quantum chemistry routines the equivalent of the human appendix. The whole problem of interupts is now in the hands of the operating system, as the data servers are distinct processes. To follow the same example as before, when the compute process p=1 needs data that turns out to reside on node 0, a request is sent to the data server p=2 to transfer information back to the compute process p=1. The compute process p=0 is completely unaware that such a transaction has occurred.

The formula for the memory required by any single node is unchanged, if p is the total number of nodes used, total on any 1 node = MEMORY + MEMDDI/p.

* * *

At present, only closed shell MP2 gradients, and the ZAPT open shell MP2 energy take advantage of the new distributed memory options. We expect to adapt other methods to use this technique of memory aggregation, but currently all other types of jobs run with MEMDDI=0 and therefore use only replicated storage. In this case the data server processes still run, but are dormant because no distributed memory access is attempted. For example, in an SCF computation (no hessian or MP2 follow on) the memory needed is on the order of the square of the basis set size, for such quantities as the orbital coefficients, density, Fock, overlap matrices, and so on. These are simply duplicated on every node in the MEMORY region.

Check runs (EXETYP=CHECK) need to run quickly, and the fastest turn around always comes on one node only. Runs which do not currently exploit MEMDDI distributed storage will formally allocate their MEMORY needs, and feel out their storage needs while skipping almost all of the real work. Since MEMORY is replicated, the amount that is needed on 1 node remains unchanged if you later do the true computation on more than 1 node.

Check jobs which involve MEMDDI storage are a little bit trickier. As noted, we want to run on only 1 node to get fast turn around. However, MEMDDI is typically a large amount of memory, and this is unlikely to be available on a single node. The solution is that the data server process does not actually allocate the MEMDDI storage, instead it just remembers what you gave as input and checks to see if this will be adequate. So, you can input MEMDDI=1000 (1000 million words is equal to 1,000 * 1,000,000 * 8 = 8 GBytes and run this check job on a computer with only 256 MB of RAM.

Of course, the actual computation will have to run on a large number of such processors. Let us continue with this example of a run requiring 8 GBytes of distributed data on 256 MB nodes. Suppose that MEMORY is 2500000 in this case (when MEMDDI is used, MEMORY is typically just a few million words). We need to reserve some memory for the operating system (16 MBytes, say) and for the GAMESS program and local storage (approx 16 MB, it is a big program, and the compute processes should be swapped into memory). Thus our hypothetical 256 MB node has 224 MB available, assuming no one else is running. The rest of the computations proceed in million/mega words, so the available memory per node is 224/8 = 28. We must choose the number of processors p to satisfy needed <= available MEMORY + MEMDDI/p <= free physical memory 2.5 + 1000/p <= 28 so this example requires p >= 39 compute processes.

One more subtle point about CHECK runs with MEMDDI is that since you are running on 1 node only, the code does not know that you wish to run the parallel MP2 algorithm instead of the sequential algorithm. You must force the CHECK job into the parallel section of the program by $system parall=.true. $End There's no harm leaving this line in for the true runs, as any job with more than one compute process is parallel regardless of the input value PARALL.

* * *

The next section deals with compilation and execution of GAMESS. If someone else has already figured these things out for you, you may skip ahead to the section that illustrates how the code's performance scales.

The purpose of this section is to describe only how the choices for low level message passing to support the DDI subroutines impact upon installation. More explicit directions for the compiling process can be found in the first two sections of this chapter, in the readme.unix file, and notes on your machine and its compilers are to befound in the IRON.DOC chapter and in the 'comp' script. This section has the best explanation available of how to execute the program.

The message traffic generated by DDI calls is sent by SHMEM on the Cray T3E, by MPI-1 on large parallel computers, and by TCP/IP sockets on networks of workstations. We cover each of these three classes of machines next.

- - -

The Cray T3E's SHMEM library affords a single process implementation of GAMESS. The T3E's message passing is contained in DDIT3E.SRC, and selecting the T3E target when compiling will use only this file and link against the SHMEM library. The 'rungms' script has a special target to permit execution using this library.

In general we expect a large vendor supplied parallel computer such as the IBM SP, SGI Origin, and large systems from companies such as Fujitsu, NEC, and Hitachi to have a MPI-1 library available. It is furthermore reasonable to assume that an expensive machine in this class has a budget sufficient to purchase the vendor's MPI library. Therefore the compute process/data server model outlined above will activate *MPI lines in the source file DDI.SRC, and link with the MPI-1 library. Each requires a special target in the 'rungms' execution script. Execution will require the vendor's "kickoff" routine to start two processes on each node, the second half of these will automatically become the data servers.

Since DDI is fairly new, we have MPI control language in the scripts 'compall', 'comp', 'lked', and 'rungms' for the IBM SP and SGI Origin only. Other machines with MPI-1 libraries should be easy to port to, needing only that you write control language for, as there is no reason to doubt the MPI-1 messaging code, since it functions on more than one machine already.

- - -

The third class of machines are technical workstations running Unix. In this category we include the IBM RS/6000, Compaq AXP (yours may say Digital on the front), Sun, HP, and SGI workstations, and also Intel-based Linux systems. These are characterized by low cost, implying that even if a vendor offers MPI-1 on these systems, the software may not have been purchased. However, all of these have the TCP/IP socket library that has been in Unix for decades now. Chances are that a vendor MPI-1 runs over sockets on this class equipment anyway, so DDI might just as well talk directly to sockets. The socket code was written with frequent reference to the original TCGMSG source code, and consequently is programmed in a similar fashion.

The DDI socket code consists of C language routines to open socket connections and transmit data through them, in the file DDISOC.C. Higher level concepts such as global communications are written in FORTRAN, as the *SOC lines in DDI.SRC. Besides these two files which link into the GAMESS executable, we need a way to fire up the compute processes and data servers. This is DDIKICK.C which is referred to as the "kickoff program".

The compiling and linking scripts 'compall', 'comp', and 'lked' have targets for each kind of workstation, as their compilers have various options. When the compiling and linking is done you should have two programs, namely ddikick.x and gamess.01.x. The latter can be run on one or more CPUs, as it is sequential if you run on one node, and parallel whenever you run on more than one. The execution script 'rungms' has a common target of 'sockets' since all six machines

we have mentioned used ddikick.x to start processes. This script has more details about how to run, but we will describe here what the arguments to the kickoff program are.

The command in 'rungms' to fire up GAMESS is

```
% ddikick.x Inputfile Exepath Exename Scratchdir \
Nhosts Hostname_0 Hostname_1 ... Hostname_N-1
```

The Inputfile name is not actually used, but it will be displayed by the 'ps' command so you can tell what is actually being run. Exepath is the name of the directory that contains the program to be executed. Exename is the name of the GAMESS executable (which might have a different "version number" than 01). The best situation is to have Exepath in an NFS mounted partition that all nodes can access, so that you have only one copy of the big GAMESS executable. However, you could carefully FTP a copy to all nodes using always exactly the same file name, such as /usr/local/bin/gamess.01.x.

Note that since only one executable name is specified, only one vendor's computers can be used at a time. This limitation arises from a lack of XDR calls in the DDI layer to convert data types from one internal representation of numeric data to another machine's.

Scratchdir is the name of a large working disk space, such as /scr/mike, in which all temporary files are placed. These files should be automatically deleted by the execution script as the job ends. If the nodes do not happen to have the same scratch area name, you can make it "feel like" they do with soft links such as "ln -s /actualname /scr". Under no circumcumstance should you make Scratchdir an NFS partition, as serious I/O happens to this directory. Ideally Scratchdir is a striped multi-disk Ultra-2-wide partition, with 9+ GBytes free space per node. However, the GAMESS output (stdout) and two supplemental ASCII output files PUNCH and IRCDATA can and probably should be sent over NFS to the user's permanent disk space on a file server. This serves the purpose of allowing the user to monitor the simulation as it runs, and gets the results to a place where it can be backed up once in a while. The files written into Scratchdir should be erased by the 'rungms' script upon normal exit.

Individual file names are set by the 'rungms' script's setenv commands. Some of the files are written to only by the master process running on node 0 (stdout and PUNCH are good examples of this), but other files are distributed across all node's Scratchdirs (scalable disk usage). The atomic integrals AOINTS is a good example of this. The rungms setenv's will define this file as xxx.F08 on node 0, where xxx is the name of the input file, and the rest of the name comes from its being FORTRAN unit 8 internally. On other nodes the file name will have the node number appended, xxx.F08.001, xxx.F08.002, and so on. Obviously, only the compute processes own disk files.

Nhosts is the number of compute processes to be run. If you want to run sequentially, just ensure Nhosts is 1. Hostname_0 is the "master node", which handles reading the one input file, and writing the one output file. This host must be the same host that is executing the 'rungms' script, or else the environment variables that define the files don't get properly accepted. Supply a total of Nhosts Hostnames. One compute process will be started on each of these (process IDs 0,1,...Nhosts-1), and then one data server will be run on each as well (for a total of 2 times Nhosts processes). If you have SMP systems, such as a four processor machine, set Nhosts=4, and repeat its Hostname 4 times.

Execution is by a direct system call if the process is to run on the host running 'rungms' and which is therefore also running 'ddikick.x'. Remote hosts are reached by the command 'rsh', so users will need to use a .rhosts file to authenticate themselves (unless your system is using some replacement for this such as Kerberos). The .rhosts file needs to be in your home directory, and looks like this:

```
si.fi.ameslab.gov  mike
```

```
ga.fi.ameslab.gov  mike
ge.fi.ameslab.gov  mike
...and so on...
```

except your user name is probably not 'mike'. Note that ddikick.x has no mechanism to support the user name on one of the machines being 'schmidt' instead of 'mike'.

Assuming that all goes well, the job will terminate orderly by each compute process telling its local data server to cease execution. Upon the successful suicide of the data server, the compute process reports to the dormant (but still running) ddikick.x that it is ready to end. When all compute processes have checked in, the kickoff program informs each that it is OK to stop, and following this ddikick.x exits.

Abnormal terminations are of course less predictable. However, when ddikick.x is informed by the system that one of its children has died, it tries to send a kill command to all its other children, and so hopefully all processes are then eliminated. However, depending on the exact circumstances in which the abnormal end occurs, the system may have a few processes left over for manual termination. If you decide that a GAMESS job should be killed, use the Unix 'kill' command to take out either the compute or data server process on the master node, or one of the 'rsh' processes that have launched GAMESS onto the remote nodes. Do not kill ddikick.x directly, instead stop any of these child processes, so that ddikick.x will terminate all the other processes for you.

Before ending this section on DDI over TCP/IP sockets on workstation class machines, we should comment on the network requirements. It is not reasonable to run jobs that use MEMDDI distributed memory on 10 megabit/second Ethernet since the bandwidth is just too small. However, if you use only the replicated MEMORY storage you should be able to get by on this old network cable. As will be shown below, a switched Fast Ethernet is capable of decent performance on such 100 megabit/second cables. Both the host adapters and the switch itself are now inexpensive. Gigabit ethernet (1000 mbit/sec) is pricy, and although the bandwidth is good, the latency remains too large.

* * *

This section describes the way in which the various quantum chemistry computations run in parallel, and shows some typical performance data. This should give you as the user some idea how many nodes can be efficiently used for various SCFTYP and RUNTYP jobs. There's a different subsection for 4 different kinds of runs, followed by a summary.

Many of the performance data you will see below were obtained on a 16 node Intel Pentium II Linux (Beowulf-type) cluster costing $49,000, of which $3,000 went into the switched Fast Ethernet component. 512 MB/node means this cluster has an aggregate memory of 8 GB. For more details, see

http://www.msg.ameslab.gov/GAMESS/page/not/written/yet

\- - -

The HF wavefunctions can be evaluated in parallel using either conventional disk storage of the integrals, or via direct recomputation of the integrals. Assuming the I/O speed of your system is good, direct SCF is *always* slower than disk storage. Some experimenting will show which is more effective on your hardware. As an example of the scaling performance of RHF, ROHF, UHF, or GVB jobs that involve only computation of the energy or its gradient, we include here a timing table from the 16 node PC cluster. The molecule is luciferin, which together with the enzyme luciferase is involved in firefly light production. The chemical formula is C11N2S2O3H8, and RHF/6-31G(d) has 294 atomic orbitals. There's no molecular symmetry.

The run is done as direct SCF because the total amount of AO integrals is 3.8 GBytes, and Linux does not permit files to exceed 2 GBytes (of course use of 2 or more nodes can be run conventional as this disk file will be distributed across all available disks). The CPU timing data is

	p=1	p=2	p=3	p=4	p=8	p=12	p=16	
1e- ints	1.6	0.8	0.6	0.4	0.3	0.3	0.1	
Huckel guess	22	18	16	14	14	12	12	
15 RHF iters	5536	2802	1891	1436	753	519	406	
properties	7.5	7.3	7.3	7.3	7.8	7.0	7.0	
1e- gradient	11.5	5.7	4.1	2.7	1.4	1.0	0.8	
2e- gradient	1339	658	437	328	105	110	83	
total CPU	6917	3491	2357	1790	941	649	509	seconds
total wall	6924	3540	2408	1820	979	696	559	seconds

Note that direct SCF should run with the wall time very close to the CPU time as there is essentially no I/O and not that much communication (MEMDDI storage is not used by this kind of run). Wall clock speedup from 1 to 16 nodes is 12.4, and for this type of run we frequently use 8, 16, or 32 nodes depending on availability.

An idea of the variation in time with basis set size can be gained from the following runs made by Johannes Grotendorst, Juelich, Germany, on a Cray T3E or Intel Paragon, using 32 nodes on either. These data were collected in about 1996, pre-DDI days, and as you can see, before all the Paragons were unplugged. The data is still representative. Each molecule is an asymmetric organic compound, computing the RHF energy and gradient, using the 6-31G(d) basis set:

			T3E	Paragon	
taxol,	1032	AOs, CPU TIME =	546.8	--	minutes
cAMP,	356	AOs, CPU TIME =	14.6	106.4	
luciferin,	294	AOs, CPU TIME =	8.9	67.2	
nicotine,	208	AOs, CPU TIME =	3.8	26.1	
thymine,	149	AOs, CPU TIME =	1.5	12.2	
alanine,	104	AOs, CPU TIME =	0.5	5.2	
glycine,	85	AOs, CPU TIME =	0.3	3.2	

If you are interested in an explanation of how the parallel SCF is implimented, see the main GAMESS paper, M.W.Schmidt, K.K.Baldridge, J.A.Boatz, S.T.Elbert, M.S.Gordon, J.H.Jensen, S.Koseki, N.Matsunaga, K.A.Nguyen, S.J.Su, T.L.Windus, M.Dupuis, J.A.Montgomery J.Comput.Chem. 14, 1347-1363(1993)

For the next type of computation, we discuss the MP2 correction. For UHF + MP2, only the second order energy can be computed, and the parallization strategy is similar to the replicated MEMORY code used by the MCSCF program. This is described below. The MCSCF + MP2 code does not run parallel, unfortunately. So here we are describing the closed shell RHF + MP2 energy or gradient, or the ROHF + ZAPT-type MP2 energy. These two types of computations make use of the MEMDDI distributed data region.

The example is a benzoquinone precursor to hongconin, a cardioprotective natural product. The formula is C11O4H10, and 6-31G(d) has 245 AOs. There are 39 valence orbitals included in the MP2 treatment, and 15 core orbitals. MEMDDI must be 156 million words, so the same type of memory computation that was used above tells us that our 512 MB/node PC cluster must

have at least three processors to aggregate the required MEMDDI. MOREAD was used to provide converged RHF orbitals, so only 3 RHF iterations are performed. The timing data are CPU and wall times (seconds) in the 1st/2nd lines:

	p=3	p=4	p=8	p=12	p=16
RHF iters	208	157	83	58	46
	214	163	91	68	55
MP2 step	8,935	6,966	3,417	2,283	1,724
	12,529	10,046	5,763	4,013	3,056
2e- grad	2,181	1,712	838	552	420
	2,490	1,981	991	677	499
total CPU	11,335	8,846	4,347	2,902	2,199
total wall	15,248	12,206	6,859	4,772	3,624

	3-->12	4-->16
CPU speedup	3.91	4.02
wall speedup	3.20	3.37

On a T3E machine with 600 MHz nodes and 256 MB/node, we should have been able to run on as few as 6 nodes, but the available data for the same calculation starts from 8:

	p=8	p=32	p=128
total CPU	3108	814	282
total wall	3154	850	340

Wall clock performance is considerably better as you would expect on a machine where very good communications exist. Wall speedup for a 4x or 16x increase in node number is 3.7 and 9.3. Larger molecules which still have some computation left at 128 nodes do better than this. We often use 64 or 128 nodes for this type of run.

As noted, the number of nodes is more influenced by a need to aggregate the necessary total MEMDDI, more than by concerns about scalability. MEMDDI is typically large for MP2 parallel runs, as it is proportional to the number of occupied orbitals squared times the number of AOs squared.

For more details on the distributed data parallel MP2 program, see
G.D.Fletcher, A.P.Rendell, P.Sherwood, Mol.Phys. 91, 431-438(1997)
G.D.Fletcher, M.W.Schmidt, M.S.Gordon, Adv.Chem.Phys. 110, 267-294 (1999)
G.D.Fletcher, M.W.Schmidt, B.M.Bode, M.S.Gordon, Comput.Phys.Commun. submitted
The latter has additional explanations of the ideas behind distributed memory programming.

- - -

The next type of computation we will consider is the analytic computation of the hessian for RHF, ROHF, or GVB wavefunctions. The current implementation of the response equations is in the MO basis, and since the solver is not parallelized, so when this stage of the computation is reached, work is done only by process 0 while the other processes sit idle. This is a sequential bottleneck. The integral transformation is parallelized according to the same strategy as described below for MCSCF jobs. Thus our early paper on analytic hessians which ran a sequential transformation no longer correctly describes the program. Thus the total scalability is better than was shown in
T.L.Windus, M.W.Schmidt, M.S.Gordon, Chem.Phys.Lett. 216, 375-379(1993)

The example we will consider is the same SbC4O2NH4 test that was used in this early paper.

We use the 3-21G* basis (110 AOs, 2 million words used). The hardware is the same PC Linux cluster. Table 1 of the reference should be:

	p=1	p=2	p=3	p=4	
1e- ints	0.15	0.09	0.09	0.07	seconds
Huckel	4.23	3.97	4.05	4.15	
2e- ints	14.84	7.26	4.89	3.68	
RHF iters	35.17	19.29	13.89	10.84	
properties	0.39	0.40	0.38	0.42	
dupl.2e- ints	N/A	14.75	14.71	14.78	
int.transf.	232.89	123.91	85.01	68.20	
1e- hess	4.47	2.84	2.04	1.21	
2e- hess	668.13	334.60	225.50	168.31	
CPHF	224.48	210.80	206.21	192.29	
total CPU	1185.0	718.1	557.0	464.2	
total wall	1188	733	600	494	

Clearly, the final response equation (CPHF) step is a sequential bottleneck, as is the fact that the orbital hessian in this step is stored entirely on the disk space of node 0. Since the integral transformation is run in replicated MEMORY rather than distributing this, and since it also needs a duplicated AO integral file be stored on every node, the code is clearly not scalable to very many processors. Typically we would not request more than 3 or 4 processors for an analytic hessian job.

As the final example, we turn to MCSCF energy/gradient runs. The parallelization of the integral transformation was done before the introduction of the distributed data concept, and hence this kind of job uses only replicated MEMORY at present. In addition, the determinant CI step is not yet converted to parallel execution, so if you run on more than one node, MCSCF jobs must use $MCSCF CISTEP=GUGA.

Our work on parallization of MCSCF was described in the paper

T.L.Windus, M.W.Schmidt, M.S.Gordon, Theoret.Chim.Acta 89, 77-88(1994).

This points out that MCSCF has many bottlenecks, of which the most important are the integral transformation, the optimization of the CI coefficients, and the optimization of the orbital coefficients. The amount of time spent in each depends on the number of atomic orbitals, and the size of the active space, and the number of filled MOs. Since this parallelization paper came out, we have added a new converger for MCSCF, namely SOSCF, and here we will show the performance of this default converger on a fairly large example. Before doing so, we point out again that our new CI optimization step over determinants is faster on one node, but it will refuse to run in parallel, so the data shown use the older GUGA CSF program.

The integral transformation step is run in replicated MEMORY, using "passes" over the occupied orbitals. The different passes involve different occupied orbitals so the work is trivially distributed across all nodes. A better description of this can be found in the reference given. Basically the same approach is currently being used during analytic hessians or UHF MP2. The method distributes CPU time fairly well, but it does not make efficient use of memory as every node must have the same memory is required to run on 1 node (order of the basis set cubed).

There are two strategies to govern the placement of AO integrals, each of which has to be available on every node. One is to store the file on the disks in a distributed fashion, and as each node reads its subset, to broadcast these on the communication channel to all other nodes. This is

AOINTS=DIST in the $TRANS or $MP2 input, and is appropriate only for machines with good communications. When disk I/O is faster than the communications, such as is likely for workstation clusters, the entire 2e- AO integral file is duplicated on every node (AOINTS=DUP). This is not scalable either in generation of integrals, or in their disk storage, but it takes pressure off the communications channel. You may want to experiment with the AOINTS keyword to see if the default for your machine is well chosen.

The example we choose is at a transition state for the water molecule assisted proton transfer in the first excited stat of 7-azaindole. The formula is C7N2H6(H2O), there are 190 active orbitals, and the active space is the 10 pi electrons in 9 pi orbitals of the azaindole portion. There are 5,292 CSFs. See Figure 6 of G.M.Chaban, M.S.Gordon J.Phys.Chem.A 103, 185-189(1999) if you are interested in this chemistry. The timing data (seconds) from our PC Linux cluster are

	p=1	p=2	p=3	p=4
dup. 2e- ints	373.2	381.2	381.4	389.9
DRT	0.4	0.4	0.4	0.4
transform.	448.0	237.4	181.3	139.1
Hamilton.	3.1	2.5	2.4	2.3
diag. H	27.4	15.1	11.1	9.1
2e- dens.	2.4	2.3	2.2	2.2
orb. update	60.1	56.7	56.2	56.4
iters 2-16	6723.3	4429.9	3554.2	2885.7
1e- grad	6.2	2.6	3.2	1.4
2e- grad	912.3	463.9	327.3	242.8
total CPU	9485.1	5599.1	4526.9	3736.1
total wall	15,703	9,537	7,763	6,192

The first iteration is broken down into its primary steps from the integral transformation to the orbital update, inclusive. Typically we would not use more than 4 to 8 processors for a parallel MCSCF job.

- - -

In summary, most ab initio computations will run in less time on more than one node. However, some things can be run only on 1 node, namely semi-empirical runs determinant based MCSCF MCQDPT2 perturbation correction to MCSCF RHF+CI gradient PCM solvation model Several steps run with little or no speedup, and thus represent sequential bottlenecks that limit scalability. They do not prevent jobs from running, but restrict the total number of nodes that can be effectively used:

HF: solution of SCF equations
HF analytic hessians: the coupled Hartree-Fock
MCSCF: orbital improvement steps
MCSCF/CI: Hamiltonian and 2e- density matrix
energy localizations: the orbital localization step
transition moments/spin-orbit: the final property step

Future versions of GAMESS will address these bottlenecks.

A short summary of the useful number of nodes (based on data like the above) would be approximately

RHF, ROHF, UHF, GVB energy and/or gradient	16-32+
analytic hessians for these	3-4
RHF + MP2 gradient, ZAPT energy	64-128+
UHF + MP2 energy	8-16

* * *

The final section of this description of DDI is a very sketchy introduction to programming. At this point, if you are interested only in using the program, you may cease reading.

DDI has subroutine calls to do ordinary message passing parallel programming. These are calls to initialize and terminate the various processes, point to point send and receive, and collective operations like global sum and the broadcast. Not necessarily every routine one would expect is included, as we just programmed what we needed in GAMESS. In addition we have calls for distributed data manipulation, which include creation and destruction of the arrays, the put, get, and accumulation operations mentioned above, and a routine to query what part of the distributed array is stored locally.

The full API for DDI is in comments in the beginning of the source file DDI.SRC, and you can then look at the individual routines to see what the calling arguments do. The source code in DDI.SRC, and calls to it from GAMESS are what serves as documentation for use of DDI at the present. Every DDI routine is a subroutine, not a function, and each begins with "DDI_" so you can easily locate all parallel constructs in GAMESS by a search for "CALL DDI_".

We don't really intend for DDI to be a general parallel programming library, rather it's a part of GAMESS. For example, we need to link a small program using DDI against some GAMESS objects to have memory management code, and so forth. These are ddi.o, ddisoc.o, unport.o, and zunix.o and maybe a timing routine.

We close with a simple (and not very useful) program that broadcasts information to all nodes. It illustrates the proper initialization and closure of the DDI library, requests replicated MEMORY but not distributed MEMDDI, and so does not illustrate distributed data programming. The idea was to keep it a one-pager.

```
      program bcast
      implicit double precision(a-h,o-z)
      parameter (maxmsg=500000)
      real tarray(2)
      common /fmcom / xx(1)
      data exetyp/8hRUN     /
c        open DDI, tell it integer word length is 32 bit
      nwdvar=2
      call ddi_pbeg(nwdvar)
c        request allocation of only replicated memory
      memrep=maxmsg
      memddi=0
      call ddi_memory(memrep,memddi,exetyp)
      call setfm(memrep)
c        start a clock so we can time this
      xstart = etime(tarray)
c        learn which of the processes I am
      call ddi_nproc(nproc,me)
      master=0
      if(me.eq.master) write(iw,9000) nproc
c        dynamically allocate a replicated array
      call valfm(loadfm)
      lbuff = loadfm + 1
      last  = lbuff  + maxmsg
      need = last - loadfm - 1
```

```
      call getfm(need)
c        fill it up with nothing but ones
      if(me.eq.master) then
         do i=1,maxmsg
            xx(lbuff-1+i) = 1.0d+00
         end do
      end if
c        send it to all the other compute processes
      call ddi_bcast(102,'F',xx(lbuff),maxmsg,master)
c        we are now done with the replicated storage
      call retfm(need)
c        so, all we've done is time a broadcast.
      xstop = etime(tarray)
      write(6,9010) me,xstop-xstart
c        close the DDI library gracefully
      istat=0
      call ddi_pend(istat)
      stop
 9000 format(1x,'running',i4,' processes.')
 9010 format(1x,'node',i4,' total job time between',
     *          ' pbeg/pend is',f7.2)
      end
```

Altering program limits

Almost all arrays in GAMESS are allocated dynamically, but some variables must be held in common as their use is ubiquitous. An example would be the common block which holds the basis set. The following Unix script, which we call 'mung', changes the PARAMETER statements that set various limitations:

```
#!/bin/csh
#       automatically change GAMESS' built-in dimensions
chdir /u3/mike/gamess/source
foreach FILE (*.src)
   set FILE=$FILE:r
   echo ===== redimensioning in $FILE =====
   echo "C 01 JAN 98 - SELECT NEW DIMENSIONS" \
             > $FILE.munged
   sed -e "/MXATM=500/s//MXATM=100/" \
       -e "/MXFRG=50/s//MXFRG=1/" \
       -e "/MXDFG=5/s//MXDFG=1/" \
       -e "/MXPT=100/s//MXPT=1/" \
       -e "/MXAOCI=768/s//MXAOCI=768/" \
       -e "/MXRT=100/s//MXRT=100/" \
       -e "/MXSP=100/s//MXSP=1/" \
       -e "/MXTS=2500/s//MXTS=1/" \
       -e "/MXSH=1000/s//MXSH=1000/" \
       -e "/MXGSH=30/s//MXGSH=30/" \
       -e "/MXGTOT=5000/s//MXGTOT=5000/" \
       $FILE.src >> $FILE.munged
   mv $FILE.munged $FILE.src
end
exit
```

In this script,

MXATM = max number of atoms
MXFRG = max number of effective fragment potentials
MXDFG = max number of different effective fragments
MXPT = max number of effective fragment points
MXAOCI= max number of basis functions in CI/MCSCF
MXRT = max number of CI roots saved by $GUGDIA
MXSP = max number of spheres (sfera) in PCM
MXTS = max number of tesserae in PCM
MXSH = max number of symmetry unique shells
MXGSH = max number of Gaussians per shell
MXGTOT= max number of symmetry unique Gaussians

The script shows how to -minimize- memory use, by a a small decrease in the number of atoms, and turning off the effective fragment and PCM dimensioning. Little can be saved by reducing the other adjustable parameters. Of course, the 'mung' script shown above could also be used to increase the dimensions...

Names of source code modules

The source code for GAMESS is divided into a number of sections, called modules, each of which does related things, and is a handy size to edit. The following is a list of the different modules, what they do, and notes on their machine dependencies.

module	description	machine dependency
ALDECI	Ames Lab determinant full CI code	1
BASECP	SBKJC and HW valence basis sets	
BASEXT	DH, MC, 6-311G extended basis sets	
BASHUZ	Huzinaga MINI/MIDI basis sets to Xe	
BASHZ2	Huzinaga MINI/MIDI basis sets Cs-Rn	
BASN21	N-21G basis sets	
BASN31	N-31G basis sets	
BASSTO	STO-NG basis sets	
BLAS	level 1 basic linear algebra subprograms	
COSMO	conductor-like screening model	
CPHF	coupled perturbed Hartree-Fock	1
CPROHF	open shell/TCSCF CPHF	1
DDI	message passing library interface code	9
DDIT3E	message passing code (used on T3E only)	9
DELOCL	delocalized coordinates	
DFT	grid-free DFT drivers	1
DFTAUX	grid-free DFT auxiliary basis integrals	
DFTINT	grid-free DFT integrals	1
DFTFUN	grid-free DFT functionals	
DMULTI	Amos' distributed multipole analysis	
DRC	dynamic reaction coordinate	
ECP	pseudopotential integrals	
ECPDER	pseudopotential derivative integrals	
ECPHW	Hay/Wadt effective core potentials	
ECPLIB	initialization code for ECP	
ECPSBK	Stevens/Basch/Krauss/Jasien/Cundari ECPs	
EIGEN	Givens-Householder, Jacobi diagonalization	
EFDRVR	fragment only calculation drivers	
EFELEC	fragment-fragment interactions	
EFGRD2	2e- integrals for EFP numerical hessian	
EFGRDA	ab initio/fragment gradient integrals	
EFGRDB	" " " " "	
EFGRDC	" " " " "	
EFINP	effective fragment potential input	
EFINTA	ab initio/fragment integrals	
EFINTB	" " " "	
EFPAUL	effective fragment Pauli repulsion	
EFPCOV	EFP style QM/MM boundary code	
FFIELD	finite field polarizabilities	
FRFMT	free format input scanner	
GAMESS	main program, single point energy and energy gradient drivers, misc.	
GRADEX	traces gradient extremals	
GRD1	one electron gradient integrals	
GRD2A	two electron gradient integrals	1
GRD2B	specialized sp gradient integrals	
GRD2C	general spdfg gradient integrals	
GUESS	initial orbital guess	

GUGDGA	Davidson CI diagonalization	1
GUGDGB	" " "	1
GUGDM	1 particle density matrix	
GUGDM2	2 particle density matrix	1
GUGDRT	distinct row table generation	
GUGEM	GUGA method energy matrix formation	1
module	description	machine dependency
GUGSRT	sort transformed integrals	1
GVB	generalized valence bond HF-SCF	1
HESS	hessian computation drivers	
HSS1A	one electron hessian integrals	
HSS1B	" " " "	
HSS2A	two electron hessian integrals	1
HSS2B	" " " "	
INPUTA	read geometry, basis, symmetry, etc.	
INPUTB	" " " "	
INPUTC	" " " "	
INT1	one electron integrals	
INT2A	two electron integrals	1
INT2B	" " "	
IOLIB	input/output routines,etc.	2
LAGRAN	CI Lagrangian matrix	1
LOCAL	various localization methods	1
LOCCD	LCD SCF localization analysis	
LOCPOL	LCD SCF polarizability analysis	
MCCAS	FOCAS/SOSCF MCSCF calculation	1
MCQDPT	multireference perturbation theory	1
MCQDWT	weights for MR-perturbation theory	
MCQUD	QUAD MCSCF calculation	1
MCSCF	FULLNR MCSCF calculation	1
MCTWO	two electron terms for FULLNR MCSCF	1
MOROKM	Morokuma energy decomposition	1
MP2	2nd order Moller-Plesset	1
MP2DDI	distributed data parallel MP2	
MPCDAT	MOPAC parameterization	
MPCGRD	MOPAC gradient	
MPCINT	MOPAC integrals	
MPCMOL	MOPAC molecule setup	
MPCMSC	miscellaneous MOPAC routines	
MTHLIB	printout, matrix math utilities	
NAMEIO	namelist I/O simulator	
ORDINT	sort atomic integrals	1
PARLEY	communicate to other programs	
PCM	Polarizable Continuum Model setup	
PCMCAV	PCM cavity creation	
PCMDER	PCM gradients	
PCMDIS	PCM dispersion energy	
PCMPOL	PCM polarizabilities	
PCMVCH	PCM repulsion and escaped charge	
PRPEL	electrostatic properties	
PRPLIB	miscellaneous properties	
PRPPOP	population properties	
QMMM	temporary dummy routines	
RESC	relativistic elim. small component	

RHFUHF	RHF, UHF, and ROHF HF-SCF	1
RXNCRD	intrinsic reaction coordinate	
RYSPOL	roots for Rys polynomials	
SCFMI	molecular interaction SCF code	
SCFLIB	HF-SCF utility routines, DIIS code	
SCRF	self consistent reaction field	
SOBRT	full Breit-Pauli spin-orbit compling	
SOFFAC	spin-orbit matrix element form factors	
SOZEFF	1e- spin-orbit coupling terms	
STATPT	geometry and transition state finder	
SURF	PES scanning	
SYMORB	orbital symmetry assignment	
SYMSLC	orbital symmetry assignment	
TDHF	time-dependent Hartree-Fock NLO	1
TRANS	partial integral transformation	1
TRFDM2	two particle density backtransform	1
TRNSTN	CI transition moments	
TRUDGE	nongradient optimization	
UNPORT	unportable, nasty code	3,4,5,6,7,8
VECTOR	vectorized version routines	10
VIBANL	normal coordinate analysis	
VSCF	anharmonic frequencies	
ZHEEV	complex matrix diagonalization	
ZMATRX	internal coordinates	

UNIX versions use the C code ZUNIX.C for dynamic memory. Most UNIX versions use DDISOC.C to talk to TCP/IP sockets, and DDIKICK.C to load GAMESS for execution.

The IBM mainframe version uses the following assembler language routines: ZDATE.ASM, ZTIME.ASM.

The machine dependencies noted above are:

1) packing/unpacking
2) OPEN/CLOSE statments
3) machine specification
4) fix total dynamic memory
5) subroutine walkback
6) error handling calls
7) timing calls
8) LOGAND function
9) message passing calls. DDI.SRC has both socket calls (*SOC) and MPI-1 calls (*MPI) programmed.
10) vector library calls

Programming Conventions

The following "rules" should be adhered to in making changes in GAMESS. These rules are important in maintaining portability, and should be strictly adhered to.

Rule 1. If there is a way to do it that works on all computers, do it that way. Commenting out statements for the different types of computers should be your last resort. If it is necessary to add lines specific to your computer, PUT IN CODE FOR ALL OTHER SUPPORTED MACHINES. Even if you don't have access to all the types of supported hardware, you can look at the other machine specific examples found in GAMESS, or ask for help from someone who does understand the various machines. If a module does not already contain some machine specific statements (see the above list) be especially reluctant to introduce dependencies.

Rule 2. a) Use IMPLICIT DOUBLE PRECISION(A-H,O-Z) specification statements throughout.

b) All floating point constants should be entered as if they were in double precision. The constants should contain a decimal point and a signed two digit exponent. A legal constant is 1.234D-02. Illegal examples are 1D+00, 5.0E+00, and 3.0D-2.

c) Double precision BLAS names are used throughout, for example DDOT instead of SDOT.

The source code activator ACTVTE will automatically convert these double precision constructs into the correct single precision expressions for machines that have 64 rather than 32 bit words.

Rule 3. FORTRAN 77 allows the use of generic functions. Thus the routine SQRT should be used in place of DSQRT, as this will automatically be given the correct precision by the compilers. Use ABS, COS, INT, etc. Your compiler manual will tell you all the generic names.

Rule 4. Every routine in GAMESS begins with a card containing the name of the module and the routine. An example is "C*MODULE xxxxxx *DECK yyyyyy". The second star is in column 18. Here, xxxxxx is the name of the module, and yyyyyy is the name of the routine. Furthermore, the individual decks yyyyyy are stored in alphabetical order. This rule is designed to make it easier for a person completely unfamiliar with GAMESS to find routines. The trade off for this is that the driver for a particular module is often found somewhere in the middle of that module.

Rule 5. Whenever a change is made to a module, this should be recorded at the top of the module. The information required is the date, initials of the person making the change, and a terse summary of the change.

Rule 6. No lower case characters, no more than 6 letter variable names, no imbedded tabs, statements must lie between columns 7 and 72, etc. In other words, old style syntax.

* * *

The next few "rules" are not adhered to in all sections of GAMESS. Nonetheless they should be followed as much as possible, whether you are writing new code, or modifying an old section.

Rule 7. Stick to the FORTRAN naming convention for integer (I-N) and floating point variables (A-H,O-Z). If you've ever worked with a program that didn't obey this, you'll understand why.

Rule 8. Always use a dynamic memory allocation routine that calls the real routine. A good name for the memory routine is to replace the last letter of the real routine with the letter M for memory.

Rule 9. All the usual good programming techniques, such as indented DO loops ending on CONTINUEs, IF-THEN-ELSE where this is clearer, 3 digit statement labels in ascending order, no three branch GO TO's, descriptive variable names, 4 digit FORMATs, etc, etc.

The next set of rules relates to coding practices which are necessary for the parallel version of GAMESS to function sensibly. They must be followed without exception!

Rule 10. All open, rewind, and close operations on sequential files must be performed with the subroutines SEQOPN, SEQREW, and SEQCLO respectively. You can find these routines in IOLIB, they are easy to use.

Rule 11. All READ and WRITE statements for the formatted files 5, 6, 7 (variables IR, IW, IP, or named files INPUT, OUTPUT, PUNCH) must be performed only by the master task. Therefore, these statements must be enclosed in "IF (MASWRK) THEN" clauses. The MASWRK variable is found in the /PAR/ common block, and is true on the master process only. This avoids duplicate output from the other processes. At the present time, all other disk files in GAMESS also obey this rule.

Rule 12. All error termination is done by means of "CALL ABRT" rather than a STOP statement. Since this subroutine never returns, it is OK to follow it with a STOP statement, as compilers may not be happy without a STOP as the final executable statment in a routine.

List of parallel broadcast identifiers

GAMESS uses DDI calls to pass messages between the parallel processes. Every message is identified by a unique number, hence the following list of how the numbers are used at present. If you need to add to these, look at the existing code and use the following numbers as guidelines to make your decision. All broadcast numbers must be between 1 and 32767.

20	Parallel timing
100 - 199	DICTNRY file reads
200 - 204	Restart info from the DICTNRY file
210 - 214	Pread
220 - 224	PKread
225	RAread
230	SQread
250 - 265	Nameio
275 - 310	Free format
325 - 329	$PROP group input
350 - 354	$VEC group input
400 - 424	$GRAD group input
425 - 449	$HESS group input
450 - 474	$DIPDR group input
475 - 499	$VIB group input
500 - 599	matrix utility routines
800 - 830	Orbital symmetry
900	ECP 1e- integrals
910	1e- integrals
920 - 975	EFP and SCRF integrals
980 - 999	property integrals
1000 - 1025	SCF wavefunctions
1030 - 1040	broadcasts in DFT
1050	Coulomb integrals
1200 - 1215	MP2
1300 - 1320	localization
1495 - 149	reserved for Jim Shoemaker
1500	One-electron gradients
1505 - 1599	EFP and SCRF gradients
1600 - 1602	Two-electron gradients
1605 - 1620	One-electron hessians
1650 - 1665	Two-electron hessians
1700 - 1750	integral transformation
1800	GUGA sorting
1850 - 1865	GUGA CI diagonalization
1900 - 1910	GUGA DM2 generation
2000 - 2010	MCSCF
2100 - 2120	coupled perturbed HF
2300 - 2399	spin-orbit jobs

Disk files used by GAMESS

These files must be defined by your control language for executing GAMESS. For example, on UNIX the "name" field shown below should be set in the environment to the actual file name to be used. Most runs will open only a subset of the files shown below, with only files 5, 6, 7, and 10 existing in every run. Only files 4, 5, 6, and 7 contain formatted data.

unit	name	contents
4	IRCDATA	archive results punched by IRC runs, restart data for numerical HESSIAN runs, summary of results for DRC.
5	INPUT	Namelist input file. This MUST be a disk file, as GAMESS rewinds this file often.
6	OUTPUT	Print output (FT06F001 on IBM mainframes) If not defined, UNIX systems will use the standard output for this file.
7	PUNCH	Punch output. A copy of the $DATA deck, orbitals for every geometry calculated, hessian matrix, normal modes from FORCE, properties output, IRC restart data, etc.
8	AOINTS	Two e- integrals in AO basis
9	MOINTS	Two e- integrals in MO basis
10	DICTNRY	Master dictionary, for contents see below.
11	DRTFILE	Distinct row table file for -CI- or -MCSCF-
12	CIVECTR	Eigenvector file for -CI- or -MCSCF-
13	CASINTS	semi-transformed ints for FOCAS/SOSCF MCSCF scratch file during spin-orbit coupling
14	CIINTS	Sorted integrals for -CI- or -MCSCF-
15	WORK15	GUGA loops for Hamiltonian diagonal; ordered two body density matrix for MCSCF; scratch storage during GUGA Davidson diag; Hessian update info during 2nd order SCF; [ia|jb] integrals during MP2 gradient
16	WORK16	GUGA loops for Hamiltonian off-diagonal; unordered GUGA DM2 matrix for MCSCF; orbital hessian during MCSCF; orbital hessian for analytic hessian CPHF; orbital hessian during MP2 gradient CPHF; two body density during MP2 gradient
17	CSFSAVE	CSF data for state to state transition runs.
18	FOCKDER	derivative Fock matrices for analytic hess
20	DASORT	Sort file for various -MCSCF- or -CI- steps; also used by SCF level DIIS
21	DFTINTS	four center overlap ints for grid-free DFT
23	JKFILE	J and K "Fock" matrices for -GVB-; Hessian update info during SOSCF MCSCF; orbital gradient and hessian for QUAD MCSCF
24	ORDINT	sorted AO integrals; integral subsets during Morokuma analysis
25	EFPIND	electric field integrals for EFP
26	PCMDATA	gradient and D-inverse data for PCM runs
27	PCMINTS	normal projections of PCM field gradients
30	DAFL30	direct access file for FOCAS MCSCF's DIIS; form factor sorting for Breit spin-orbit files 50-63 are used primarily for MCQDPT runs. files 51-54 are also used during spin-orbit runs.
50	MCQD50	Direct access file for MC-QDPT, its contents are documented in source code.

51	MCQD51	One-body coupling constants <I/Eij/J> for CAS-CI and other routines
52	MCQD52	One-body coupling constants for perturb.
53	MCQD53	One-body coupling constants extracted from MCQD52
54	MCQD54	One-body coupling constants extracted further from MCQD52
55	MCQD55	Sorted 2-e integrals
56	MCQD56	Half transformed 2-e integral
57	MCQD57	Sorted half transformed 2-e integral of the (ii/aa) type
58	MCQD58	Sorted half transformed 2-e integral of the (ei/aa) type
59	MCQD59	2-e integral in MO basis of the (ii/ii), (ei/ii), (ei/ei) types
60	MCQD60	2-e integral in MO basis arranged for perturbation calculations
61	MCQD61	One-body coupling constants between state and CSF <Alpha/Eij/J>
62	MCQD62	Two-body coupling constants between state and CSF <Alpha/Eij,kl/J>
63	MCQD63	canonical Fock orbitals (FORMATTED)
64	MCQD64	Spin functions and orbital configuration functions (FORMATTED)

Contents of the direct access file 'DICTNRY'

1. Atomic coordinates
2. various energy quantities in /ENRGYS/
3. Gradient vector
4. Hessian (force constant) matrix

5-6. not used

7. PTR - symmetry transformation for p orbitals
8. DTR - symmetry transformation for d orbitals
9. FTR - symmetry transformation for f orbitals
10. GTR - symmetry transformation for g orbitals
11. Bare nucleus Hamiltonian integrals
12. Overlap integrals
13. Kinetic energy integrals
14. Alpha Fock matrix (current)
15. Alpha orbitals
16. Alpha density matrix
17. Alpha energies or occupation numbers
18. Beta Fock matrix (current)
19. Beta orbitals
20. Beta density matrix
21. Beta energies or occupation numbers
22. Error function interpolation table
23. Old alpha Fock matrix
24. Older alpha Fock matrix
25. Oldest alpha Fock matrix
26. Old beta Fock matrix
27. Older beta Fock matrix
28. Oldest beta Fock matrix
29. Vib 0 gradient for FORCE runs
30. Vib 0 alpha orbitals in FORCE
31. Vib 0 beta orbitals in FORCE
32. Vib 0 alpha density matrix in FORCE
33. Vib 0 beta density matrix in FORCE
34. dipole derivative tensor in FORCE.
35. frozen core Fock operator
36. Lagrangian multipliers
37. floating point part of common block /OPTGRD/

int 38. integer part of common block /OPTGRD/
39. ZMAT of input internal coords
int 40. IZMAT of input internal coords
41. B matrix of redundant internal coords
42. not used.
43. Force constant matrix in internal coordinates.
44. SALC transformation
45. symmetry adapted Q matrix
46. S matrix for symmetry coordinates
47. ZMAT for symmetry internal coords
int 48. IZMAT for symmetry internal coords
49. B matrix
50. B inverse matrix
51. overlap matrix in Lowdin basis, temp Fock matrix storage for ROHF
52. genuine MOPAC overlap matrix
53. MOPAC repulsion integrals
54. exchange integrals for screening
55. orbital gradient during SOSCF MCSCF
56. orbital displacement during SOSCF MCSCF
57. orbital hessian during SOSCF MCSCF
58. Reserved for Pradipta
59. Coulomb integrals in Ruedenberg localizations
60. exchange integrals in Ruedenberg localizations
61. temp MO storage for GVB and ROHF-MP2
62. temp density for GVB
63. dS/dx matrix for hessians
64. dS/dy matrix for hessians
65. dS/dz matrix for hessians
66. derivative hamiltonian for OS-TCSCF hessians
67. partially formed EG and EH for hessians
68. MCSCF first order density in MO basis
69. alpha Lowdin populations
70. beta Lowdin populations
71. alpha orbitals during localization
72. beta orbitals during localization
73. alpha localization transformation
74. beta localization transformation
75. fitted EFP interfragment repulsion values
76 -77. not used
78. "Erep derivative" matrix associated with F-a terms
79. "Erep derivative" matrix associated with S-a terms
80. EFP 1-e Fock matrix including induced dipole terms
81. not used
82. MO-based Fock matrix without any EFP contributions
83. LMO centroids of charge
84. d/dx dipole velocity integrals
85. d/dy dipole velocity integrals
86. d/dz dipole velocity integrals
87. unmodified h matrix during SCRF or EFP
88. not used
89. EFP multipole contribution to one e- Fock matrix
90. ECP coefficients
int 91. ECP labels
92. ECP coefficients

int 93. ECP labels
94. bare nucleus Hamiltonian during FFIELD runs
95. x dipole integrals, in AO basis
96. y dipole integrals, in AO basis
97. z dipole integrals, in AO basis
98. former coords for Schlegel geometry search
99. former gradients for Schlegel geometry search
100. not used

records 101-248 are used for NLO properties

101. U'x(0)	149. U''xx(-2w;w,w)	200. UM''xx(-w;w,0)
102. y	150. xy	201. xy
103. z	151. xz	202. xz
104. G'x(0)	152. yy	203. yz
105. y	153. yz	204. yy
106. z	154. zz	205. yz
107. U'x(w)	155. G''xx(-2w;w,w)	206. zx
108. y	156. xy	207. zy
109. z	157. xz	208. zz
110. G'x(w)	158. yy	209. U''xx(0;w,-w)
111. y	159. yz	210. xy
112. z	160. zz	211. xz
113. U'x(2w)	161. e''xx(-2w;w,w)	212. yz
114. y	162. xy	213. yy
115. z	163. xz	214. yz
116. G'x(2w)	164. yy	215. zx
117. y	165. yz	216. zy
118. z	166. zz	217. zz
119. U'x(3w)	167. UM''xx(-2w;w,w)	218. G''xx(0;w,-w)
120. y	168. xy	219. xy
121. z	169. xz	220. xz
122. G'x(3w)	170. yy	221. yz
123. y	171. yz	222. yy
124. z	172. zz	223. yz
125. U''xx(0)	173. U''xx(-w;w,0)	224. zx
126. xy	174. xy	225. zy
127. xz	175. xz	226. zz
128. yy	176. yz	227. e''xx(0;w,-w)
129. yz	177. yy	228. xy
130. zz	178. yz	229. xz
131. G''xx(0)	179. zx	230. yz
132. xy	180. zy	231. yy
133. xz	181. zz	232. yz
134. yy	182. G''xx(-w;w,0)	233. zx
135. yz	183. xy	234. zy
136. zz	184. xz	235. zz
137. e''xx(0)	185. yz	236. UM''xx(0;w,-w)
138. xy	186. yy	237. xy
139. xz	187. yz	238. xz
140. yy	188. zx	239. yz
141. yz	189. zy	240. yy
142. zz	190. zz	241. yz
143. UM''xx(0)	191. e''xx(-w;w,0)	242. zx

144. xy
145. xz
146. yy
147. yz
148. zz

192. xy
193. xz
194. yz
195. yy
196. yz
197. zx
198. zy
199. zz

243. zy
244. zz

245. old NLO Fock matrix
246. older NLO Fock matrix
247. oldest NLO Fock matrix
249. not used
250. transition density matrix in AO basis
251. static polarizability tensor alpha
252. X dipole integrals in MO basis
253. Y dipole integrals in MO basis
254. Z dipole integrals in MO basis
255. alpha MO symmetry labels
256. beta MO symmetry labels
257-261. reserved for Cheol Choi
262-279. not used
280. Zero field LMOs during numerical polarizability
281. Alpha zero field dens. during num. polarizability
282. Beta zero field dens. during num. polarizability
283. zero field Fock matrix. during num. polarizability
284-289. not used
290-299. reserved for Alex Granovsky
300. Z-vector during MP2 gradient
301. Pocc during MP2 gradient
302. Pvir during MP2 gradient
303. Wai during MP2 gradient
304. Lagrangian Lai during MP2 or CI gradient
305. Wocc during MP2 gradient
306. Wvir during MP2 gradient
307. P(MP2)-P(RHF) during MP2 gradient
308. SCF density during MP2 gradient
309. energy weighted density during MP2 gradient
311. Supermolecule h during Morokuma
312. Supermolecule S during Morokuma
313. Monomer 1 orbitals during Morokuma
314. Monomer 2 orbitals during Morokuma
315. combined monomer orbitals during Morokuma
316. nonorthogonal SCF orbitals during SCF-MI
317. unzeroed Fock matrix when MOs are frozen
318. MOREAD orbitals when MOs are frozen
319. bare Hamiltonian without EFP contribution
320. MCSCF active orbital density
321. MCSCF DIIS error matrix
322. MCSCF orbital rotation indices
323. Hamiltonian matrix during QUAD MCSCF
324. MO symmetry labels during MCSCF
330. CEL matrix during PCM
331. VEF matrix during PCM

332. QEFF matrix during PCM
333. ELD matrix during PCM
340. DFT alpha Fock matrix
341. DFT beta Fock matrix
342. DFT screening integrals
343. DFT: V aux basis only
344. DFT density gradient d/dx integrals
345. DFT density gradient d/dy integrals
346. DFT density gradient d/dz integrals
347. DFT M[D] alpha density resolution in aux basis
348. DFT M[D] beta density resolution in aux basis
349. DFT orbital description
350. overlap of true and auxiliary DFT basis
351. previous iteration DFT alpha density
352. previous iteration DFT beta density
353. DFT screening matrix (true and aux basis)
354. DFT screening integrals (aux basis only)
360-369. reserved for Rob Bell
370. left transformation for pVp
371. right transformation for pVp
370. basis A (large component) during NESC
371. basis B (small component) during NESC
372. difference basis set A-B1 during NESC
373. basis N (rel. normalized large component)
374. basis B1 (small component) during NESC
375. charges of non-relativistic atoms in NESC
376. common nuclear charges for all NESC basis
377. common coordinates for all NESC basis
378. common exponent values for all NESC basis
372. left transformation for V during RESC
373. right transformation for V during RESC
374. 2T, T is kinetic energy integrals during RESC
375. pVp integrals during RESC
376. V integrals during RESC
377. Sd, overlap eigenvalues during RESC
378. V, overlap eigenvectors during RESC
379. Lz integrals
380. reserved for Ly integrals.
381. reserved for Lx integrals.
382. X, AO orthogonalisation matrix during RESC
383. Td, eigenvalues of 2T during RESC
384. U, eigenvectors of kinetic energy during RESC

In order to correctly pass data between different machine types when running in parallel, it is required that a DAF record must contain only floating point values, or only integer values. No logical or Hollerith data may be stored. The final calling argument to DAWRIT and DAREAD must be 0 or 1 to indicate floating point or integer values are involved. The records containing integers are so marked in the list below.

Physical record 1 (containing the DAF directory) is written whenever a new record is added to the file. This is invisible to the programmer. The numbers shown above are "logical record numbers", and are the only thing that the programmer need be concerned with.