NASA-CR-205011

Final Report

Contract NAS8-38609 D.O. 177

OPAD-EDIFIS Real-Time Processing

prepared for

NASA

by

Dr. Constantine Katsinis

University of Alabama in Huntsville

Electrical and Computer Engineering

June 1997

ABSTRACT

The Optical Plume Anomaly Detection (OPAD) detects engine hardware degradation of flight vehicles through identification and quantification of elemental species found in the plume by analyzing the plume emission spectra in a real-time mode. Real-time performance of OPAD relies on extensive software which must report metal amounts in the plume faster than once every 0.5 sec.

OPAD software previously written by NASA scientists performed most necessary functions at speeds which were far below what is needed for real-time operation. The research presented in this report improved the execution speed of the software by optimizing the code without changing the algorithms and converting it into a parallelized form which is executed in a shared-memory multiprocessor system. The resulting code was subjected to extensive timing analysis. The report also provides suggestions for further performance improvement by a) identifying areas of algorithm optimization, b) recommending commercially available multiprocessor architectures and operating systems to support real-time execution and c) presenting an initial study of fault-tolerance requirements.

0. COPY OF TASKS AND DELIVERABLES STATEMENT

Tasks

1. Computational Module Code Conversion.

Two modules, the SPECTRA code and Neural Network code, shall be converted from IDL to C language programs to produce code which is optimized for speed of execution and to allow the software to be integrated with a real-time operating system (most support C but not IDL). Other modules shall be converted from IDL to C language as necessary to support the full implementation of SPECTRA and Neural Network code.

2. Code Enhancement.

The resulting C code along with the Optimization module Fortran code shall be analyzed extensively to improve the algorithms and enhance the execution speed. After conversion to multitasking form to take advantage of all available parallelism within the software, the multitasking code shall be executed in a single CPU workstation and extensive experiments shall be performed to accurately establish the speed and performance of its components. After any necessary modifications, the multitasking code shall be ported to a multiprocessor system (3-6 processors) currently in operation at UAH and additional experiments shall be performed to verify earlier predictions of speed enhancement.

3. Architecture/Real-Time Operating System Study.

In parallel, an extensive study shall be performed on commercially available multiprocessor architectures and supporting real-time operating systems (RTOS) to select and recommend an architecture-RTOS combination with sufficient processing power and ease of use to allow real-time execution of combined software.

4. Fault-Tolerance Requirements Study.

An initial study shall be performed to assess the fault-tolerance requirements of the system recommended in Task 3. This study shall produce recommendations on the type of hardware and software enhancements that will be required to achieve various levels of fault-tolerance.

Deliverables

1. Prototype multitasking code derived from Task I and Task 2. Code shall include SPECTRA and Neural Network modules and other code necessary for full interface and implementation among modules used during the performance of all tasks described herein.

2. A final report shall provide a) complete descriptions of all algorithm and code enhancements which were included in the final parallel version of the software, b) results of all timing experiments which characterize the software performance, c) detailed recommendations of a multiprocessor architecture and real-time operating system, and d) preliminary recommendations on fault-tolerance requirements and possible enhancements.

1. INTRODUCTION

The NASA OPAD spectrometer system relies heavily on extensive software which repetitively extracts spectral information from the engine plume and reports the amounts of metals which are present in the plume. New data from the spectrograph is generated at a rate of once every 0.5 sec or faster. All processing must be completed within this period of time to maintain real-time performance. The software in the OPAD system performs this function by solving the inverse problem. It uses physics-based computational models which receive amounts of metals as inputs to produce the spectral data that would have been observed, had the same metal amounts been present in the engine plume. During the experiment, for every spectrum that is observed, an initial approximation is performed using neural networks to establish an initial metal composition which approximates as accurately as possible the real one. Then, using optimization techniques, the SPECTRA code is repetitively used to produce a fit to the data, by adjusting the metal input amounts until the produced spectrum matches the observed one to within a given level of tolerance.

The original version of the SPECTRA code was written in IDL, a language which facilitated analysis and visualization of results but did not seek to optimize execution speed. In addition, the iterative nature of the processing further contributes to a relatively long period of time to execute the software in a modern single-processor workstation.

The initial SPECTRA code performs a number of functions to calculate the spectrum. The initialization phase establishes basic parameter values for all metals of interest. In the input phase a file is read containing the desired concentration and broadening parameter values for the metals. Then for each element, the program reads a database file and extracts the lines for the element that are in the region of the wavelength being investigated, and then, for each line a procedure is called to compute emission due to this line. Following the calculation of the spectrum, the code generates the instrument response function and convolves it with the spectrum to produce two final arrays called the thick and the thin spectra.

To improve the execution speed, the code was translated into the C language and was modified to implement the following: 1) maintain all necessary database information in memory, so that disk access is completely avoided after the initialization phase, 2) keep in memory (cache) partial results produced by previous calculations, and 3) replace several algorithms which performed specific processing tasks within the code with improved versions designed for higher execution speed.

The resulting C version of SPECTRA was converted to a parallel version to allow further increases in execution speed. The code can be compiled in two different ways to produce two parallel forms of SPECTRA which can be executed in two different but compatible environments. The first form is multiprocessing code executing in a standard UNIX environment. It can easily be ported to other workstations allowing other researchers to experiment with the code. The second form is multiprocessor code, where different processes are actually executed on different processors. This form was developed for execution on the shared-memory multiprocessor (six M68030 processors, VxWorks real-time operating system) in operation at UAH. It allows the user to probe individual processors and perform accurate measurements on the timing and progress of the code executing on

each processor.

The multiprocessor software has been written in such a way that conversion to a different parallel environment will require a small effort, mostly adjusting the form of system calls. In fact, the first parallel form of SPECTRA (for the UNIX multitasking environment) is simply a derivative of the second (multiprocessor) form. This was accomplished by concentrating all the system specific functions in separate files and using defined variables in makefiles to specify which functions are compiled for each version. An example of such a file is parallel.c which is shown in Figure 1-1.

```
#ifdef VXWORKS
#include "parallel_vx.c"
#endif
#ifdef UNIX
#include "parallel_sun.c"
#endif
```

Figure 1-1. File parallel.c

The makefiles for the two forms mentioned above appear in Appendix A.

This report concentrates on the multiprocessor form of SPECTRA. The processors are separated in two groups. The first group contains one processor, called the coordinator. The second group contains the remaining processors, called the workers. Each worker processor executes a copy of the same process, which repetitively receives input values for the work assigned to it and calculates the contribution to the spectrum. The coordinator processor controls the activities of the worker processors. Work can be assigned to the worker processors either statically or dynamically. Since the research reported here was more concerned with the timing optimization of the SPECTRA code, work assignment in a rather static fashion was adequate. Future versions will take advantage of more sophisticated scheduling procedures which dynamically distribute the work optimally over all worker processors. The coordinator processor may calculate the convolution of the spectrum with the instrument response. It also executes the fitting (optimization) code which drives the operation of the system by specifying the input values.

The purpose of the fitting code is to treat the absolute value of the difference between the actual and the calculated spectra as a function of independent variables (the concentration and broadening parameter values of the metals of interest) and attempt to minimize this function by selecting the proper values of the these variables. Our studies have shown that as the value of any independent variable diverges from the correct one, the absolute difference of the spectra increases monotonically, and therefore the local minimum is the global one. This research has examined two different types of fitting code. One is the downhill simplex method in multiple dimensions to transform a multi-dimensional geometrical simplex through a sequence of reflections in such a way that a local minimum is approached. The second optimization is using the CFSQP V 2.4 code, available from the University of Maryland, which is designed to solve constrained nonlinear optimization problems.

2. The Host Multiprocessor System

After initial conversion to the C language and some optimization in a conventional single-processor computer, futher development and timing analysis of the SPECTRA code took place using the multiprocessor system which we have assembled at UAH. To allow easy porting of the code to other systems, only common features of the UAH multiprocessor where used.

The multiprocessor system used to test the multitasking version of SPECTRA consists of a SUN workstation, a VMEbus chassis (which can accomodate up to 12 VME boards), and a set of six (and possibly more) MVME147 single board computers (SBC). Each board is connected via Ethernet to the SUN workstation that contains the files accessed by the boards. The components of the MVME147 boards include: an MC68030 microprocessor, a Floating-point Coprocessor (MC68882), Dual-ported DRAM (4 Mbytes), an Ethernet transceiver interface, and four 32-pin ROM sockets. Each SBC contains a unique number (a processor number) within its ROM, which allows the same program to be developed for a group of SBCs with the capability to take different actions for each individual SBC as necessary. Figure 2-1 shows the system organization.

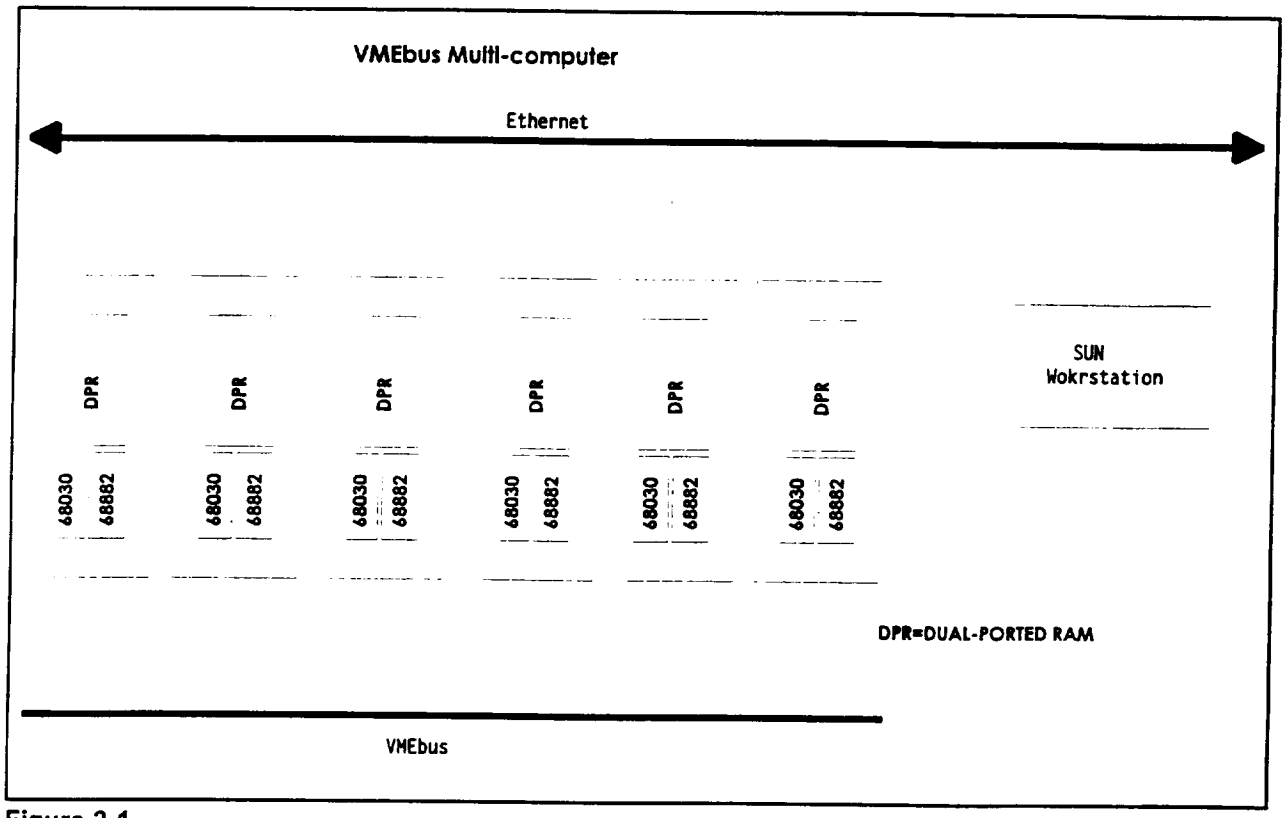

Figure 2-1.

The whole system operates under the control of VxWorks and the UNIX operating system. VxWorks is a real-time operating system acting as a partner to UNIX. The two operating systems cooperate by allowing the program development (editing, compiling,

linking and storing of real-time code) to occur on the UNIX system and the actual execution and debugging of the software to occur on the VxWorks system. Once the development cycle is complete, the software can run standalone either in ROM or disk-based or can continue to be downloaded from the network.

The VxWorks bootstrap code is placed in ROM. When the SBCs power-up, they perform some initialization of the on-board hardware, log on to the SUN system and download the VxWorks operating system. It is possible to specify a shell script that is to be executed after the boards have finished booting up. The shell script contains a list of commands exactly as the user would enter them at the command prompt of the terminal attached to the SBC. Shell scripts are used to run the software on both the coordinator and worker processors. This prevents the user from repeatedly having to type in the command with all of the command line arguments and file names in all of the SBCs every time the software is executed.

The present form of the software contains the additional functionality which allows the user to conveniently specify repetitive executions of the software (possibly different versions) with different input parameter values. These features will continue to be very useful in the development of the software and are described in detail in this report.

The use of shell scripts provides the means to completely automate the running of all versions of the software with differing number of processors. Two types of scripts are used, one for the coordinator and one for all the workers. SBC processor numbers are used to cause a program to take different actions when necessary. The two types are very similar, with the major difference being that the coordinator script causes all processors to be reset at the end of the software execution in preparation for the next execution. The scripts call programs which read data files (called "job description" files) with information specifying which executable code to load into the processors and which parameter values must be passed to the code at the beginning of the execution.

A job description file contains the command lines to be passed to the software for each test. The shell script runs a program which examines the job description file, removes the next job and invokes the SPECTRA software with the retrieved command line parameters. When the software terminates, the coordiantor script causes all the processors to be reset. When the SBCs finish rebooting, they examine the job description file and ran the next test. This continues until the job description file has been emptied, at which point the boards wait for input from the command line. Detailed descriptions of the shell scripts, job description files, and job retrieval programs are shown in Appendix B.

The description above illustrates why the CreateWorkerProcess() for VxWorks does nothing. VxWorks task spawning functions do not provide a way to specify a particular CPU that the task (or process) is to be executed on. To simplify matters, Worker processes are created when the system boots up by commands from a shell script. The Worker does not get ahead of the Coordinator because it must wait for the Coordinator to create the shared data structures before the Workers can open them.

VxWorks supports shared-memory objects between processes running on separate processors. The memory to be used for all of the shared memory objects is allocated from the Coordinator's dual-ported DRAM. The Coordinator handles the creating of all shared data structures while the workers merely open the data structures for use. The calls to open the shared data structures are blocking. The worker processors perform some initial-

ization and then attempt to open the shared message queues. If the Coordinator has not created the message queues, the worker process blocks until the message queues are created. This is true for all of the shared data structures.

The library functions which provide the VxWorks system calls for the shared memory objects are placed in the files parallel_vx.c and parallel_vx.h.

A description of the system calls used in this library is placed in Appendix C.

3. Software Description

3.1. General Overview

The code has been written in such a way that it can be executed on the SUN workstations (UNIX) or on the VMEbus single board computers (VxWorks) without modification. There are two makefiles that handle the compilation of the code: makefile.vx and makefile.sun (these files are shown in Appendix A). In order to run the code on a specific architecture, the user can simply recompile the code using the appropriate makefile.

Initialization: vxmain, main, spectra, assign, load, reade, readt
Spectrum: spectrb, worker, voi, wav, abs, nrutil
Multiprocessing: parallel, parallel_vx, parallel_sun
Timing functions: event, event_vx, event1
Other functions: func, mtest, spfit, amoeba

Figure 3-1. C files that make up SPECTRA

3.1.1. Operating system abstraction

Figure 3-1 shows the C files that make up the SPECTRA code. The lowest level operating system specific routines, used to create, delete and access shared data structures, are kept in separate library files (parallel_vx.c, parallel_sun.c, event_vx, event_SGI etc....). The interface to the library routines is kept consistent by using the same function names and number of parameters for all operating systems. This approach keeps the operating system specific calls hidden at the lowest level of the code and greatly simplifies the task of porting the software to new operating systems. The user can simply create a version of the library files that handles the system calls of the new operating system and create a makefile that defines the appropriate flags (-DVXWORKS, -DUNIX for example) which will cause the proper library files to be used when building the executable.

In UNIX, shared memory objects are identified by keys (unique integers) and in VxWorks shared memory objects are identified by names (strings). In order to access the shared object the user must know the object's ID (whether it is a key or a name). These details are hidden from the user by implementing the functions with respect to a particular data structure. An example of this is the set of functions that provide access to the FIXED_GLOBALS structure: CreateFxGlobals(), OpenFxGlobals(), DeleteFxGlobals(). The structure's ID (key or name) is used only inside the library file. This approach makes the code easy to port to other operating systems and much easier to work with from the programmer's perspective.

When using VxWorks, the worker processors start executing the worker code on boot up. In UNIX, the worker processes are forked as part of the initialization in specinit(). For this reason, CreateWorkerProcess() actually forks the worker processes when using the UNIX system but does nothing when using VxWorks.

3.1.2. Communication description

Communication between the coordinator and the workers is implemented by messages. There are four message types currently being used: INIT, READY, START, and FINISH.

The main purpose of the INIT message is to let the workers know that the Fx_globals and datn_array data structures have been initialized. The number of processors

that will be used for this run is sent in the INIT message. The INIT message is sent to all processors in the system. The worker process decides whether he should participate in this run by comparing his processor number with the number of processors in the INIT message. If the worker's processor number is greater than the number of processors participating, he exits. For this reason, the INIT message is sent to all processors in the system but the remaining messages are sent only to participating processors.

The READY message is sent from each worker process to the coordinator process to indicate that the worker process has finished it's initialization and is ready to begin the calculations.

The START message is sent to the worker processes after new input has been received (new metal concentrations and/or spectrap) and the related global data structures have been updated. The START message contains parameters that are related to the spectrap structure. The values include llimit, range, temp, length and press. Since all of the workers required these parameters, is was faster to send them as part of the message than to have the workers trying to read them from shared memory at the same time.

The FINISH message sent from the worker processes to the coordinator contains the processor number of the worker process that sent the message. The coordinator waits until it receives messages from all of the processors participating in the calculations before continuing. Since the spectra calculations can be performed a number of times, the worker processes complete there calculations and wait for the next message. The coordinator process sends a FINISH message to the workers when there are no more calculations to do. This signals the worker processes to exit.

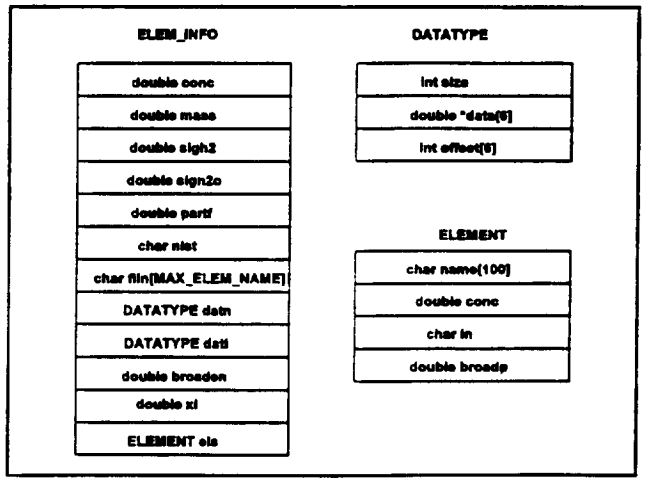

Figure 3-2. ELEM_INFO Structure

3.1.3. Program operation

The input for the program is in the form of data files containing parameters for the elements involved in the spectrum calculations. The information is kept in data structures in shared memory so it can be accessed by all of processors. After the initialization of the global variables is complete, each of the processors is assigned the element(s) on which it will perform the calculations. The processors can be assigned single elements, groups of elements or portions of elements in order to achieve effective load balancing. After each processor has finished determining the local spectrum for the assigned element(s), a global sum operation is done to combine the separate local spectra into the final spectrum containing the contributions made by all of the elements. A series of checks is performed on the data and then the global spectrum is convolved with the instrument response to provide the final spectrum output.

3.2. Shared memory data structures

All information between processes is passed through data structures and message

queues in shared memory. The definitions of these data structures are located in the common.h header file, parts of which are shown in Appendix Y for reference.

Array elem_info contains structures of type ELEM_INFO, one for each element. Figure 3-2 shows the information of this and related structures. Each structure contains the element parameters from the element.lst file and an array of pointers to arrays containing the element database information. Initialization of this structure is performed by the functions load_spec1() [load.c] and read_elements() [reade.c]. The element database is written into shared memory by the function read_bin() [readt.c] and the pointers to these arrays are updated in the elem_info structure. In addition the element concentrations and broadening parameters are stored here and updated from the comp.lst file by the load_spec2() [load.c] function.

Array Fx_globals contains a structure of type FIXED_GLOBALS of global read_only variables, mainly command options and the array of structures containing the element assignments for the workers. Figure 3-3 shows the information of this and related structures.

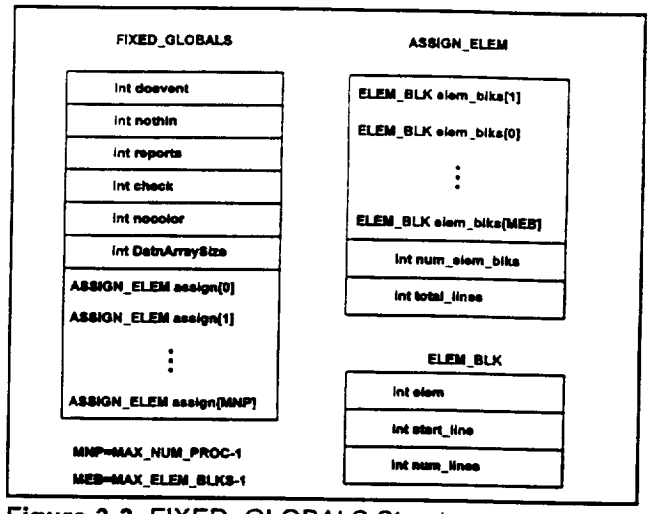

Figure 3-3. FIXED_GLOBALS Structure

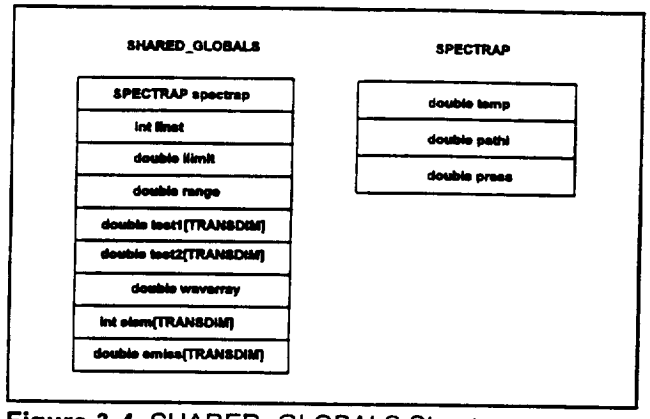

Figure 3-4. SHARED_GLOBALS Structure

Array Sh_globals contains a structure of type SHARED_GLOBALS which groups together a number dynamic and read-write variables. These variables are obtained from the comp.lst file in the function load_spec2() [load.c]. Figure 3-4 shows the information of this and related structures.

Other global variables include spectrum[], spectrum2[] which contain the thick and thin global spectrum values, and the data arrays used to hold the element database. Figure 3-5 shows the complete organization of the shared memory.

In addition, shared memory contains data structures used for multitasking such as semaphores for spectra, data arrays, and the linst variable, and message queues for the coordinator and worker processes. Figure 3-6 shows the organization

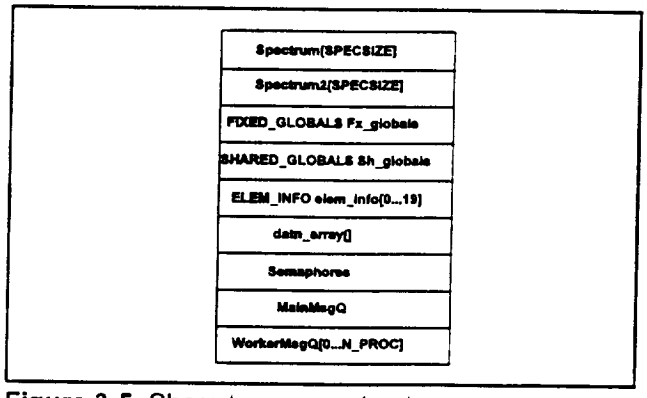

Figure 3-5. Shared memory structure

of message queues and their functionality in shared memory.

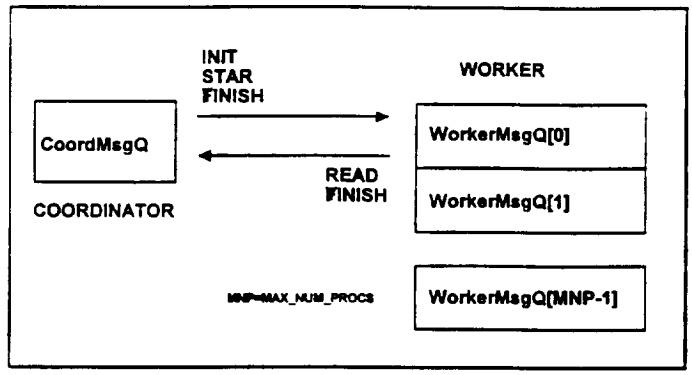

Figure 3-6. Coordinator and Worker Queues

3.3. Coordinator functions

The individual files and functions of the coordinator are described below. Figure 3-7 shows a basic flowchart of the coordinator operation.

Vxmain.c

The file vxmain.c is used when compiling for VxWorks. It was necessary because VxWorks tasks are spawned using the function sp(funcname, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9). The main function in vxmain.c (called start()) receives a string containing the command line to be passed to the main() function in main.c. Start() handles converting the string into the typical main(int argc, char *argv[]) arguments used in UNIX. This approach keeps the main() function in main.c the same for both operating systems. The only difference in the two versions is that in VxWorks a command line string is passed to start() which then calls the main(int argc, char *argv[]) with the typical arguments.

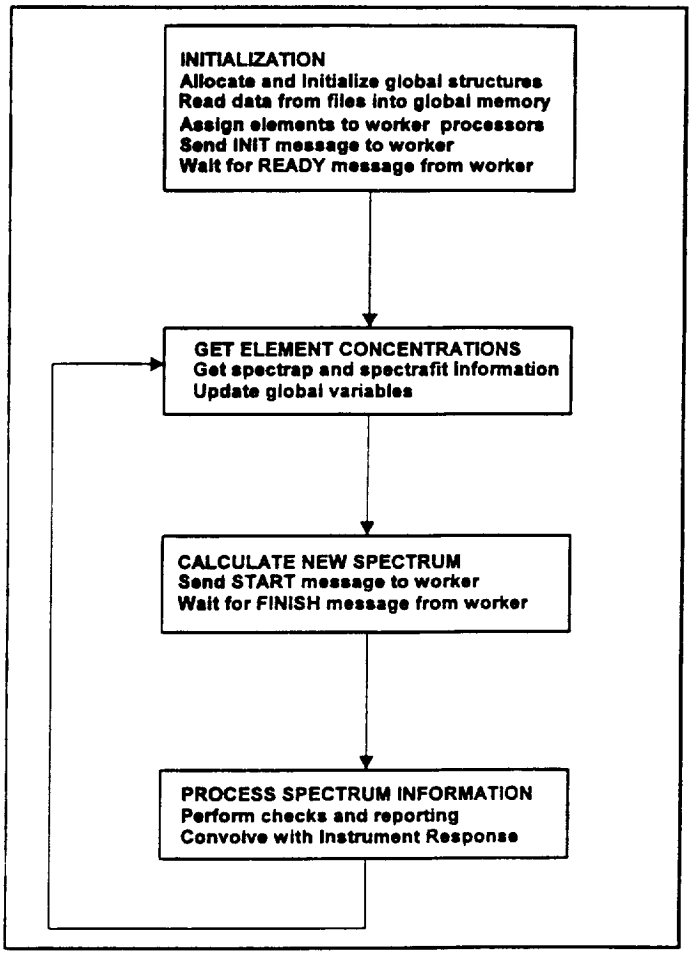

Figure 3-7. Coordinator Processor Flowchart

Main.c

The function main() sets the global command parameters according to the command line arguments. A set of timing routines is provided in the file event.c. Function event_i() initializes the data structures used by the timing routines. Initialization of the global data structures that are used in the determination of the spectrum is handled in the routines specinit() [spectra.c] and load_spec2() [load.c]. The determination of the spectrum and the convolution with the instrument response takes place in the function spectra() [spectrb.c]. In order to run the code multiple times with different element concentrations, the function mtest() [mtest.c] is used. Mtest() contains a loop which reads the new comp.lst file, calls load_spec2() and then calls spectra(). Event_p() handles the printing of the timing information gathered during

the program's execution.

Input data files

The input to the spectrum software is located in the following files:

element.lst: provides partition data for each of the elements. The entries for each of the elements contains the name of the file which contains the element database and other parameters associated with the element (partf, nist, mass, sigH2, sigH2O).

.bn3: The original files containing the element database are ASCII files. The information from these files were read and then written to binary files. The binary files contain the same name as the ASCII files but with a different extension, (.bn3). The element database files contain data organized in 6 parallel arrays containing wavelength values and limits. All 6 arrays of a single metal have the same number of data points but the number of data points in the arrays can differ from element to element.

comp.lst: This file contains the concentration of the various metals as well as Spectra parameters (spectrap data structure) and Spectra fitting parameters such as wavelength shift and response magnification parameters (spectrafit data structure).

Data from the element.lst and element database files does not change for different iterations of the program. Element concentration data from the comp.lst file can change with each iteration of the program.

Spectra.c: specinit()

The routine specinit() allocates and initializes the global data structures in shared memory. It then creates message queues for the worker processors and distributes the elements among them. In the UNIX version of the code, the processes are created at this point but in the VxWorks version, the processes are created at system startup and the CreateMinorProcess() function does nothing. At this point an INIT message is sent to each of the processors indicating that the global variables have been set up and are ready to be retrieved. The coordinator process then waits for the worker processes to send a READY message indicating that they have finished their initialization and are ready to perform the calculations.

Load.c: load_spec2()

The function load_spec2() reads the data from the comp.lst file and stores the conc and broadp values for each of the elements in the elem_info structure. Load_spec2() also fills in the spectrap and spectrafit structures with the parameters in comp.lst.

assign.c: AssignByElem(), AssignByLines()

These functions handle the distribution of elements to the worker processes. The assignment decisions are based on the number of lines that will be used from each of the elements. The number of lines used from the elements is determined by the parameters in the spectrap structure which is updated from the comp.lst file.

Currently two methods of assignment have been examined. Assigning whole elements and assigning portions of elements. The difference in the number of lines used for the different metals is so drastic, 1 or 2 lines for some and over 200 lines for other metals, that balancing the amount of work performed by each of the processors is very difficult. This topic will be discussed further in the section on timing analysis.

If the spectrap structure is the same for all of the iterations then this method of assignment is acceptable. If the spectrap structure changes with each iteration, the number of lines that are considered from the elements will change every time. In order to maintain

a good load balance between the worker processes, new assignments would need to be made each iteration. This will affect the execution time and will be examined further in the section on timing analysis.

Spectrb.c: spectra()

Spectra() contains the major portion of the code executed by the coordinator process. After initializing variables with the information received from load_spec2(), the coordinator sends a START message to the worker processes. The worker processes calculate their portions of the spectrum and then combine the results to obtain the final spectrum. After sending the START message, the coordinator waits until receiving the FINISH message from the worker processes.

The coordinator performs some checks to verify an assumption made in the absorp() [abs.c] function and then performs a convolution with the instrument response function to obtain the final output (specthick and specthin). The function doresp() [spectra.c] determines the response function to be used in the convolution and depends on the variables associated with determining the range of wavelengths to be considered in the spectrum (spectrap and spectrafit structures).

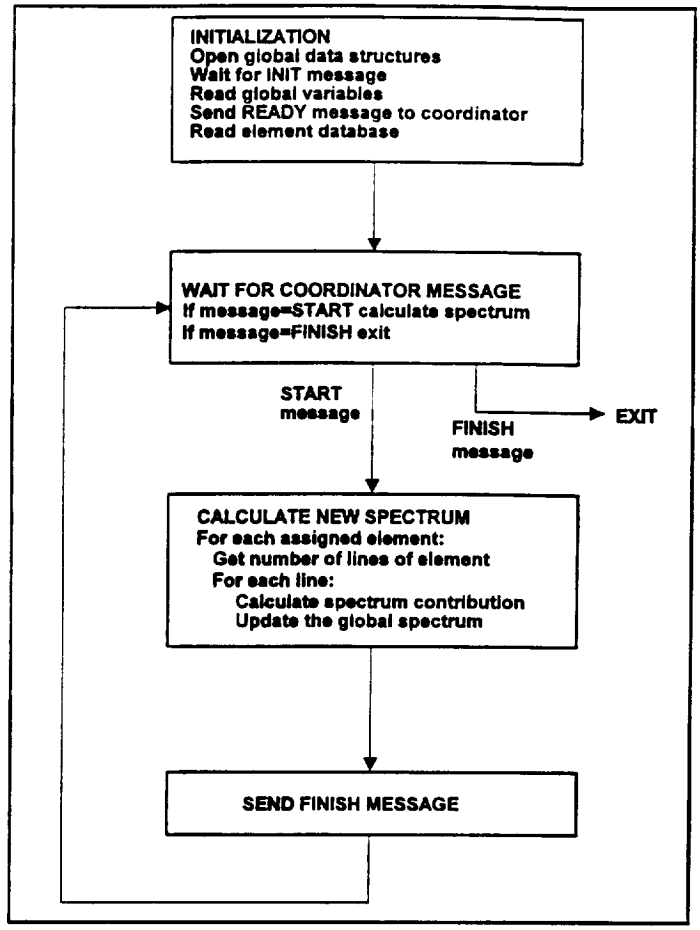

Figure 3-8. Worker Processors Flowchart

At this point the convolution is performed. The original version of the convolution resulted in 83% of its calculations involving a data point of the spectrum that was zero. A modification of the algorithm resulting in a 50% decrease in the time used to perform the convolution by only considering non-zero spectrum entries.

```
double absorp ()
 {
  range = line(drange,broaden,shape);
  for (i=0;i<range;i++) kline[i]=mag*shape[i]/dop;
  sum = 0;
  for (i = 0; i < range; i++)
   sum += 1.0 - exp(-1*kline[i]*length/100.0);
  return(tauint=sum * 0.1 * dop);
 }

line (drange,a,shape)
double drange, a;
FLOAT *shape;
 {
  tot = 0.;
  for (i = 0; i < range; i++)
   tot += (shape[i] = voigt(a,x[i]));
  tot *= 0.1;
  for (i = 0; i < range; i++) shape[i] /= tot;
  return(range);
 }
```

Figure 3-9. Original absorp() code

3.4. Worker files and functions

The individual functions of the worker are described below. Figure 3-8 shows the flowchart of the worker operation.

Worker.c

The worker process must first open the global message queues and data structures cre-

ated by the coordinator process. When the process receives the INIT message, it makes a local copy of the static global variables and data structures. At this point the worker process sends the coordinator a READY message and then goes into a loop receiving and processing messages from the coordinator until a FINISH message is received at which point the worker exits.

When the worker process receives the START message it makes a local copy of the dynamic global variables and then proceeds to perform the calculations for the metals that were assigned to it. The main body of the calculation consists of an outer loop that is done for each element and an inner loop that is done for each line in the specified range of the element.

First the element database is scanned to retrieve the data lines that fall within the specified range. The function that calculates the emission value for the line is absorp() [abs.c]. This function was responsible for 99% of the execution time of the original version of the code. Several optimizations of this function have drastically reduced the time taken by this function. Specific details of these optimizations will be discussed in timing analysis section of this report.

```
double absorp()
 {
  myj = (range - 1) / 2;
  new2 = (myj + 1) - qgaus(klinefunc,0.,myj);
  tauint = 2. * new2 * 0.1 * dop;
  return(tauint);
 }
double qgaus(func,a,b)
double a,b,double (*func)();
{
 int j;
 double xr,xm,dx,s;
 static double x[]={0.0,0.1488743389,0.4333953941,
  0.6794095682,0.8650633666,0.97390652};
 static double w[]={0.0,0.2955242247,0.2692667193,
  0.2190863625,0.1494513491,0.06667134};
 xm=0.5*(b+a);
 xr=0.5*(b-a);
 s=0;
 for (j=1;j<=5;j++)
  {
   dx=xr*x[j];
   s += w[j]*((*func)(xm+dx)+(*func)(xm-dx));
  }
 return s *= xr;
}
double klinefunc(x)
double x;
 {
  int il, i2;
  double temp, dx;
  return(exp(-multip * shape[(int) x]));
 }
```

Figure 3-10. New integration in absorp

The emission values are stored in a local spectrum array. When all of the elements assigned to the worker have been processed, the global spectrum is updated with the values from the local spectrums from each of the processors. The worker then sends a FINISH message to the coordinator and waits for the next message.

3.5. Algorithm and Code Enhancements

3.5.1. Absorp Algorithm Modifications

An analysis of the original version of the software indicated that almost all of the execution time was due to the absorp() routine. There were two major areas that required changes: the determination of the shape[] array for each line and the integration. The modifications made to these sections of the code are described below. The orginal code, shown in Figure 3-9, called function line(), which returned data in array shape[], which was used to creat array kline[]. An exponential function of this array was then integrated, using an approximation to the Simpson's-rule method, to produce a value of

```
Calculation from original algorithm:
iwav = userealwav[jj] FACTOR + subset[i];

subset[i] is determined by the following loop
for(i = 0; i < respsize; i++)
 subset[i] = i - respsize/2.0;

Placing this value directly into the calculation:
iwav = userealwav[jj] FACTOR + i - respsize/2.0;

Yielding the final equation:
i = iwav + respsize/2.0 - userealwav[jj] FACTOR;

Value of i is used as the index to the response
array.
```

Figure 3-11.

variable tauint which was returned.

Experiments revealed that the data returned by line() in array shape[] is used more than once, through successive calls to function absorp(). A software cache was added in function line() which maintained previously calculated data, so that when the function is called with previously used arguments, the data is simply retieved rather than calculated again.

Further experiments revealed that the integration in the original version of absorp() resulted in a very large number of calls to function exp(), which is relatively expensive computationally. It was replaced by new code implementing the Gauss-Legendre integration algorithm. A description of this algorithm can be found in the book "Numerical Recipes in C", by W.H. Press et.al., Cambridge University Press. Figure 3-10 show the relevant segments of the new code. The use of the function pointer, func, which points always to function klinefunc() was left unchanged because it is convenient and intuitive.

3.5.2. Modifications to the Convolution Algorithm

The original convolution algorithm consisted of nested loops which performed calculations on a range of wavelengths (loop iterations = respsize) for every value in the userealwav array (loop iterations = realwavsize). Of the total loop iterations (which is respsize * realwavsize) approximately 83% of the calculations were useless because they involved a spectrum value of zero.

The purpose of the original algorithm was to use data from a range of wavelengths surrounding the wavelength contained in userealwav[jj] to calculate the value for specthick[jj]. This means that any one wavelength affects several elements of specthick[]. The number of specthick[] elements affected by a specific wavelength depends on the range of wavelengths taken into account (respsize) and the number of wavelengths between consecutive elements in userealwav[].

```
iwav_diff = (userealwav[realwavsize - 1]
          - userealwav[0]) / realwavsize
start = userealwav[0]FACTOR +subset[0];
end = userealwav[realwavsize - 1] FACTOR
     + subset[respsize - 1];
overlap = (respsize/2.0) / (iwav_diff FACTOR);
for (iwav =start ; iwav < end; iwav++)
 {
  if(spectrum[iwav] != 0.0)
   {
    temp2 = spectrum[iwav] * dimsize;
    /* calculate the index corresponding to this wavelength */
    my_jj = (((double)(iwav)/10.0)
          - userealwav[0]) / iwav_diff;
    /*
    fill in all values of specthick[] that are affected by
    this wavelength the number of values depends on
    1) step-size of userealwav 2) respsize
    */
    for(jj  = my_jj - overlap; jj <= my_jj + overlap; jj++)
     {
      /* don't try to access values out of our range */
      if((jj >= 0) && (jj < realwavsize))
       {
        /*
        get the index of the response function that should be
        used for each of these points: this calculation was
        generated by substituting the equation for subset[]
        into the equation from the original algorithm used
        to generate iwav
        */
        i = (double)iwav + (double)respsize/2.0
          - userealwav[jj] FACTOR + 1;
        if((i >= 0) && (i < respsize))
         specthick[jj] += temp2 * response[i];
       }
     }
   }
 }
```

Figure 3-12.

The new algorithm approaches the problem in another way. It scans the wavelengths in the range indicated by userealwav[] and searches for wavelengths where spectrum[wavelength] is not equal to zero. Once a contributing wavelength is found, the index of userealwav[] containing the range of wavelengths into which this one would fall is determined. This index will be the "central index" of specthick[] that will be affected by

this wavelength. The respsize and step-size of userealwav[] are used to determine surrounding specthick[] elements that will be affected by this wavelength. The next step is to determine which response[] element should be used to calculate each of the specthick[] elements. The equation used to get the response element index was taken from the original algorithm's calculation to determine iwav, as shown in Figure 3-11. This algorithm makes two assumptions: 1) the userealwav[] array has a constant step-size or wavelength range, and 2) the userealwav[] array contains elements in increasing order. The code used to implement this algorithm is shown in Figure 3-12.

4. Timing results and analysis

4.1. Evolution of the Spectrum Software Modifications

This section describes the timing analysis of the software through different stages of modification. The figures in this section of the report use event numbers to identify the portions of the code for which the execution time was measured. Table 1 shows the event numbers and their corresponding code sections in the coordinator program. Similarly, Table 2 shows the events of the worker program.

Table 1. Coordinator Events

Event	Code Sections
1	spectra()
2	initialization of spectrum[] and spectrum2[]
3	major FOR loop (for each metal)
4	save spectrum to file
6	do_resp()
7	convolution
8	save specthick and specthin to a file
9	code executed when nocolor = 0
10	save oplot colors to file (*.emw)
11	scale specthick and specthin done if source > 0
12	code executed when check = 1
13	code executed when cross = 1

Table 2. Worker Events

Event	Code Sections
0	worker main body
1	all code not accounted for by the other event
3	scan element data base to get lines in the appropriate range
4	absorp()
6	update local and global copies of spectrum[] and spectrum2[]
9	code executed when nocolor = 0
12	code executed when check = 1

As improvements to the code were made, significant differences in the execution time were observed when some of the options were selected (check = 1, for example). For this reasons, two versions of the code are discussed, the real-time version of the code which is the version used during the actual measurement phase and the debug version which is used in the test phase. The debug version is characterized by variables check and nocolor having a value of 1. The value of SPECSIZE was reduced to half of the original

value due to the memory limitations on the single board computers used in the experiments.

4.1.1. Version 1

This is the starting point for the soft ware modification. It has all of the original algorithms from the IDL Version of the code translated into the C language. The chart below shows the execution time for the events described above. Figure 4-1 shows that practically all of the execution time occurs during the calculation of the spectrum and more specifically inside the absorp() routine. This clearly was the place to start looking for areas of the code that could be optimized.

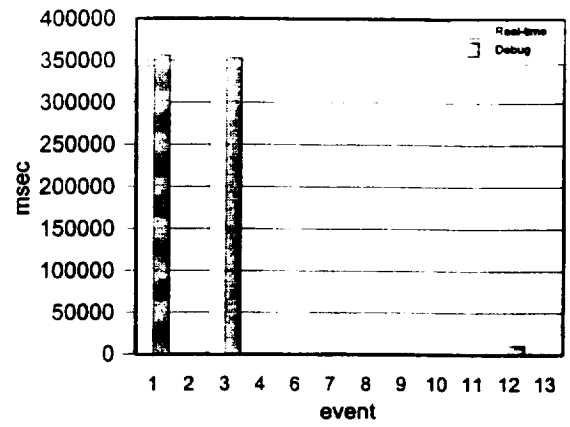

Figure 4-1. Version 1: Spectra Timing

4.1.2. Version 2

Figure 4-2 shows the speed im provement after the implementation of the software cache to hold a number of the shape functions (line() in abs.c) so they could be reused instead of regenerated. A discussion of this modification can be found in the Algorithm Description section of this report. This modification allowed the real-time Version to run 79% faster and the debug Version to run 77% faster.

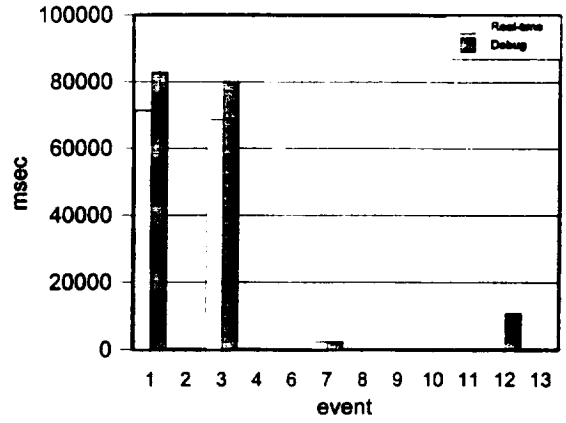

Figure 4-2. Version 2: Spectra timing

4.1.3. Version 3

Figure 4-3 shows the speed im provement after the implementation of the Gauss-Legendre integration algorithm over the shape array for every line. A change in the algorithm that determines the value for the integration resulted in a 92% performance increase for the real time Version. The debug Version exhibited a 30% performance increase. The reason for this is the array assignment that occurs for every line when check = 1.

From this point on, this report concentrates on the real-time Version of the code since that is the Version that must

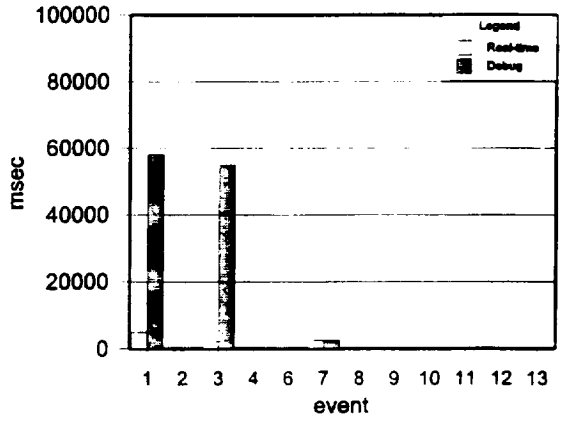

Figure 4-3. Version 3: Spectra timing

ultimately meet the timing requirements. Figure 4-4 shows the timing of the real-time SPECTRA. It is apparent that the convolution in function absorp() (event number 7) is making a significant contribution to the execution time.

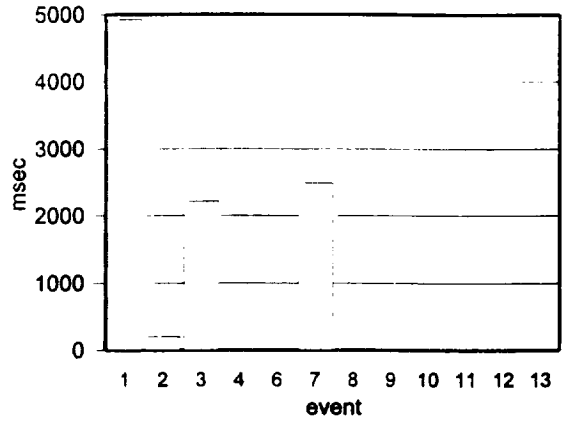

Figure 4-4. Version 3: Real-time Spectra

4.5. Version 4

Experiments have demonstrated that most of the calculations of the convolution in function absorp() were involving zero values. A restructuring of this part of the code made it more efficient by only considering non-zero values. A more detailed explanation of this modification can be found in the Algorithm Description section of this report. Figure 4-5 shows the timing with the new convolution algorithm. This modification provided an additional 28% increase in performance.

The remaining modifications involved converting the software to a multi-tasking Version and distributing the spectrum calculations (specifically the major FOR loop done for each of the elements) across multiple processors. In this configuration there is one coordinator and the remaining processors are the workers. The coordinator receives the metal concentrations, updated global variables and signales the workers to begin the calculation of the spectrum.

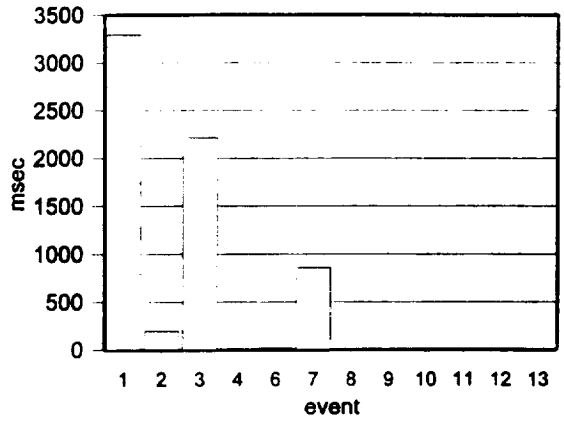

Figure 4-5. Version 4: Real-time Spectra

4.6. Version 5

In this Version of the code, the elements are allocated to each of the processors for analysis. The time required to calculate the spectrum is directly related to the number of lines used from the elements. In order to efficiently use all of the processors, it is important to assign the elements so that each of the processors ends up with the same number of lines (or as close as possible). Figure 4-6 shows the execution times for the spectra function

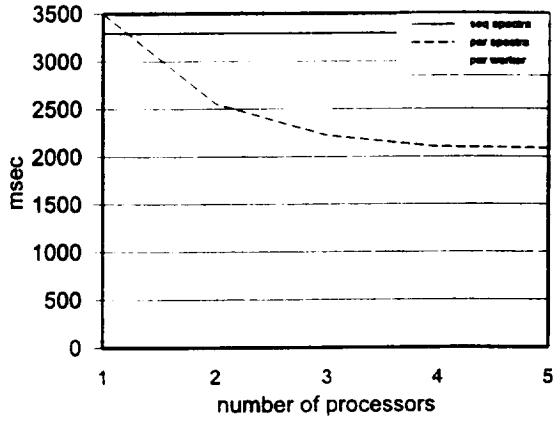

Figure 4-6. Version 5: Real-time Spectra

and for the portion of the spectra() function that is performed in parallel by the worker processors. The flattening of the curve between 4 and 5 processors is apparent.

Figure 4-7 shows a breakdown of the execution times for code sections in the spectra() function. Each of the data series represents the time for that event for the sequential Version and the multitasking Version for 1,2,3,4, and 5 processors. It is apparent that event 3 (the code executed by the worker processors) shows the same execution time for both 4-processor and 5-processor systems. The reason for this is the approach of distributing entire elements to the processors.

For the wavelength range indicated by the spectrap structure in the comp.lst file, the elements have a widely differing number of lines, some have less that 10 while others have close to 300. The uneven distribution of the number of lines per processor prevents a good load balancing for more that 4 processors since in each of these cases, the largest elements (iron: 271 lines and molybdenum: 220 lines) are allocated separately to a single processor each and the remaining processors finish their elements and remain idle while iron and molybdenum are still being analyzed. To further improve the performance, it is necessary to divide the larger elements between several processors to provide better load balancing and eliminate any idle time.

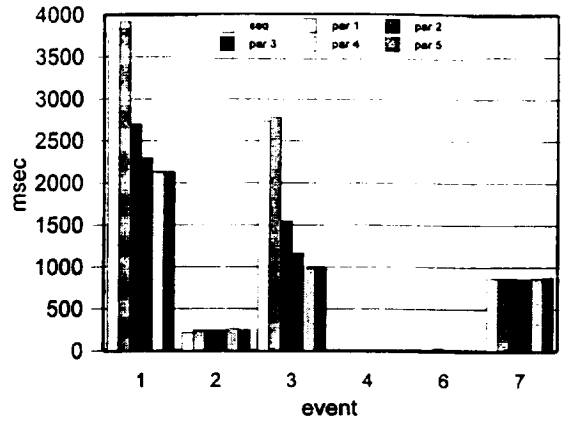

Figure 4-7. Version 5: Multiprocessor Spectra

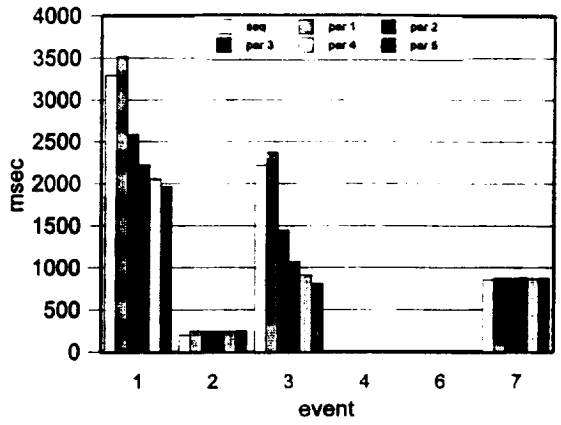

Figure 4-8. Version 6: Multiprocessor Spectra

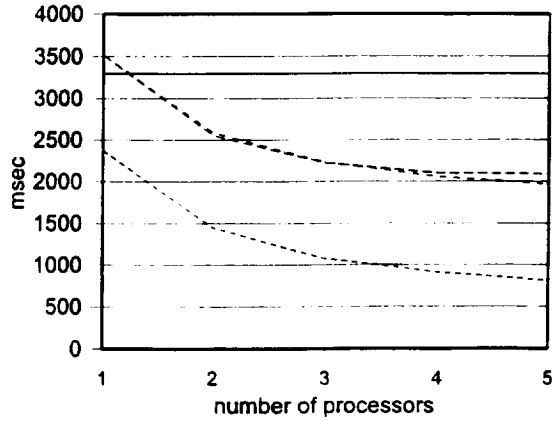

Figure 4-9. Comparison of versions 5 and 6

4.1.6. Version 6

This Version of the code allows an element to be divided between several processors in an attempt to improve the load balancing. Figures 4-8 and 4-9 show the timing breakdown for 5 processors and a comparison between Version 5 and 6 for the execution time for SPECTRA. In this Version, an increase in performance for

more that 4 processors can be seen.

The amount of speed up that can be gained by adding more processors to the system depends on the interaction between the different components of the worker code. Some components increase with increasing number of processors and some components decrease with increasing number of processors while others remain the same regardless of the number of processors. Figure 4-10 shows a breakdown of the worker components.

The most significant components are described below. In each figure, the x-axis represents the number of processors and each of the data series is the time it took to perform the code for that event in each of the processors, referred to as vme02, vme03, vme04, vme05, vme06 (vme01 is the coordinator). Figure 4-11 shows the timing for event 1 which is associated with the code sections not included by Events 3, 4 and 6. The code sections included in Event 1 that were significant are shown below.

Figure 4-12 shows event 1.3 which is the time taken to update the linst variable. This is significant because it requires taking a semaphore, updating a location in shared memory and releasing the semaphore. This is done for every line.

Figure 4-13 shows event 1.5 which is the portion of code that calculates the variables this_emiss and this_othin from the wavelength and the value returned from the absorp() routine. This portion of the code is strictly computational and is executed once for every line. It should decrease as the number of processors increases.

Figure 4-14 shows event 3 which is the part of the code where the element database is scanned to determine which lines are in the appropriate range to be used for the calculations. This portion of

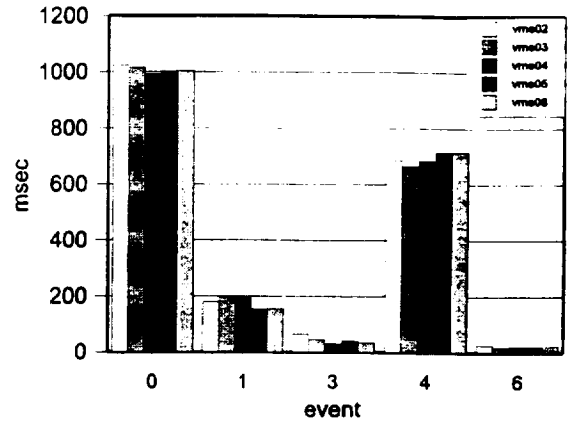

Figure 4-10. Version 6: workers

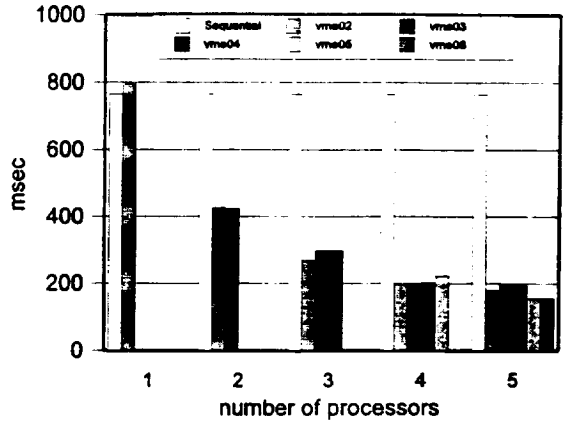

Figure 4-11. Version 6: Event 1

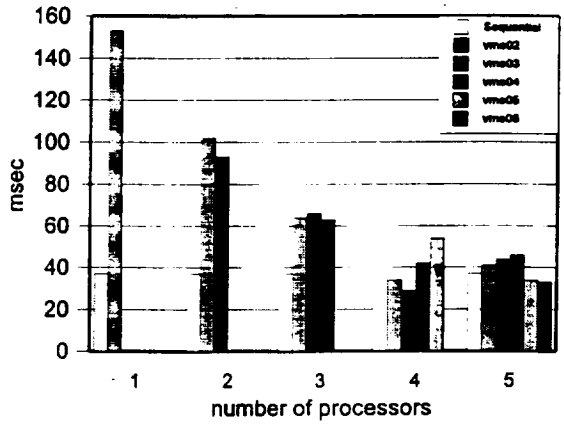

Figure 4-12. Version 6: Event #1,3

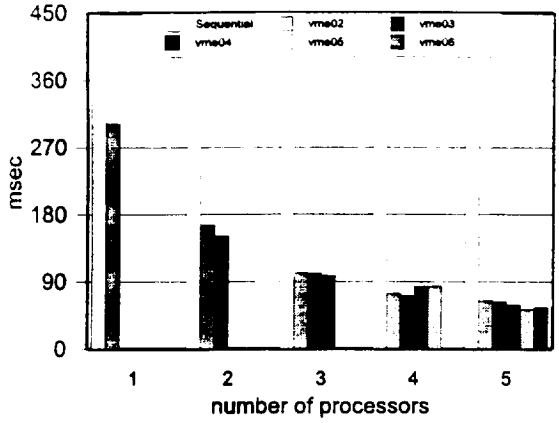

Figure 4-13. Version 6: Event #1,5

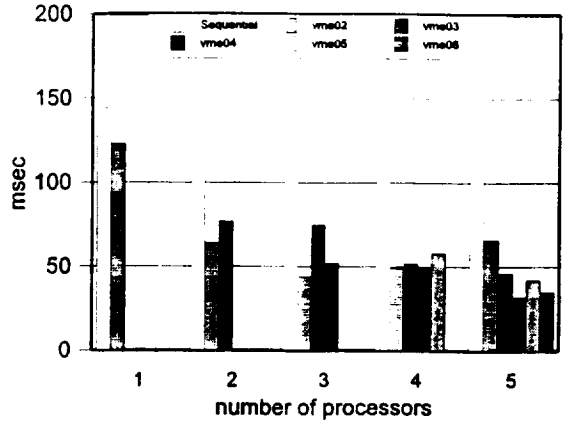

Figure 4-14. Version 6: event 3

the code must be done every time the spectrap structure is updated since the fields in this structure affect the range of wavelength that will be examined. The time it takes to perform the scanning is determined by the number of data points in the element s database and the number of elements for which the scanning must be done. In Version 5, an entire element is assigned to a processor so only one processor performs the scan for a particular element. In Version 6, an element can be distributed among several processors requiring that each processor perform the scan for that element.

Event #6 is the time required for the code sections dealing with updating the local and global spectra. Figure 4-15 shows event 6,1 which is the time spent updating the local spectrum. Figure 4-16 shows event 6,2 which is the time spent updating the global spectrum. The time for Event #6,1 will decrease as the number of processors increases since it is related to the number of lines. The time for Event #6,2 will increase as the number of processors increases because this step must be done serially. Each processor takes a semaphore, updates the global spectrum for each of its non-zero local spectrum val-

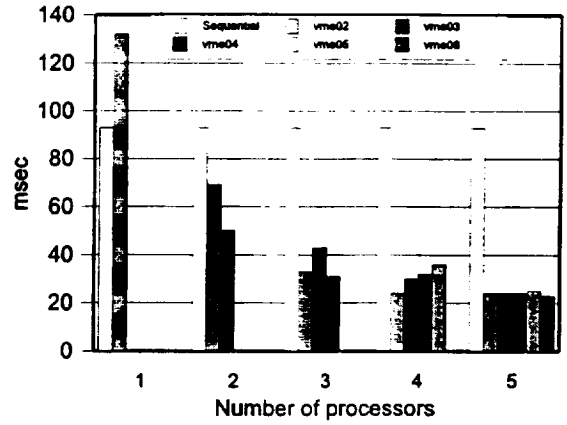

Figure 4-15. Version 6: event 6,1

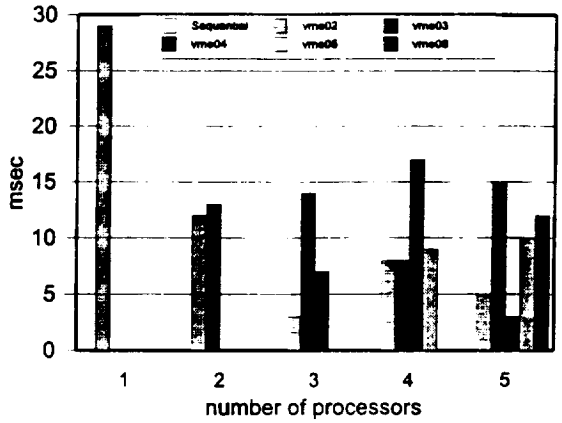

Figure 4-16. Version6: event 6,2

ues, and then releases the semaphore. Updating the global spectrum takes longer when the load balance is good because all of the processors are trying to update the spectrum at the same time and therefore must wait for each other.

Figure 4-17 shows event 4 which is the time used by the absorp() routine. It is called once for every line in the element. For an increasing number of processors, the absorp() function decreases, but not as much as would be expected.

In order to understand why the execution time for Event #4 does not drop as much as one would expect, the following diagrams show the execution times for the major components of the absorp() routine. Figure 4-18 shows event #4,1, the time required to calculate the range variable. Figure 4-19 shows event #4,2, the time required to perform the integration. Figure 4-20 shows event #4,3, the time required to obtain the shape function to be used for the line. Clearly the problem is with the time that is takes to obtain the shape function. This part of code was optimized using the software cache. The optimization prevents the shape function from being generated for each line. The problem is that it

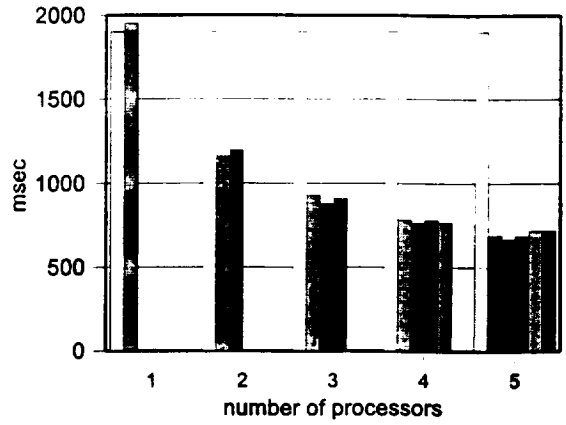

Figure 4-17. Version 6: event 4

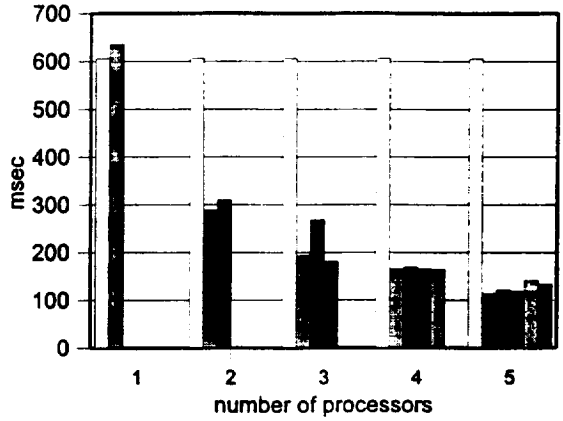

Figure 4-18. Version 6: event 4,1

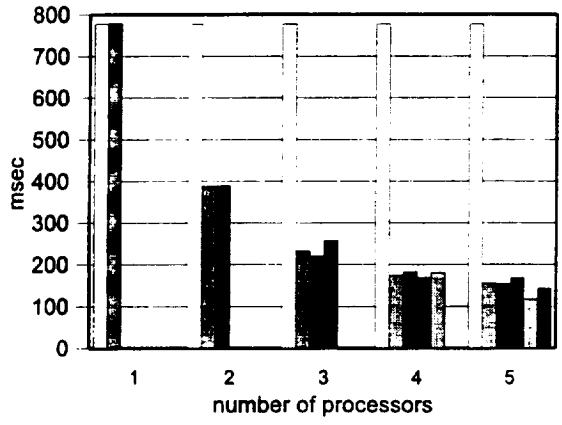

Figure 4-19. Version 6: event 4,2

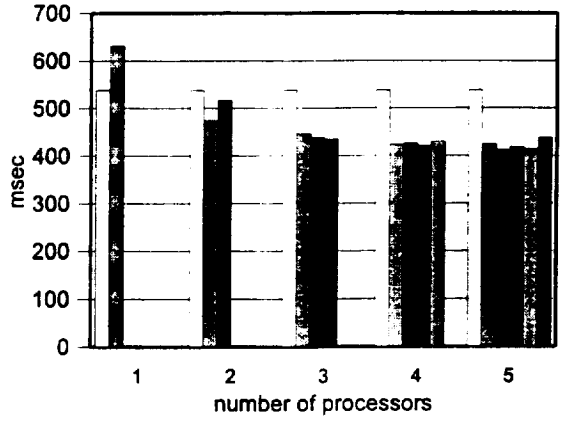

Figure 4-20. Version 6: event 4,3

takes close to 300 ms to generate the shape function. This automatically puts a bound on the amount of speedup that is possible regardless of the number of processors that are used. If the spectrap structure parameters are such that more that one shape function must be generated, the results would drastically affect the overall execution time of the program.

4.2. Observations and recommendations

4.2.1. Convolution

The time required to perform the convolution was reduced by the new algorithm, but it stills is a significant contributor to the overall execution time of the spectra() function. Additional speedup could be achieved by parallelizing the convolution code.

4.2.2 Element assignments and database scanning

The changes that should be made for the next Version of the software depend on which input parameters are expected to change each time a new spectrum is generated. The parameters in the spectrap structure affect the wavelength range that is being analyzed. If these values do not change for a particular test run. then the code for Event #3 can be moved to the initialization part of the program. This will save a lot of the overhead being observed in the worker process. It also allows the assignment of elements (or lines) to be done once at initialization time.

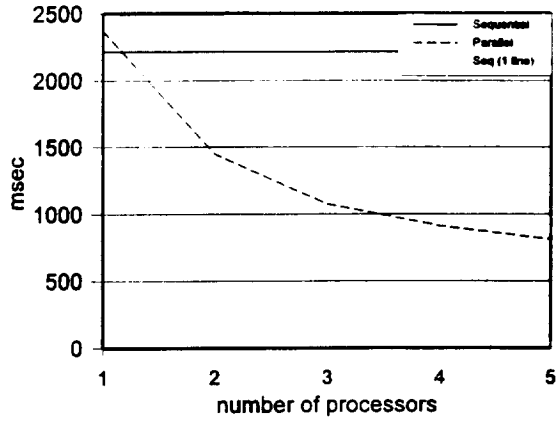

Figure 4-21. Version 6: Performance limits

If the spectrap structure is expected to change each time a new spectrum is generated, then not only must Event #3 remain in the worker part of the code, but the elements (or lines) will have to be reassigned each time. In the experiments performed above, the assignments were done at initialization. A possible approach for saving time should the spectrap structure change for each generation of the spectrum would be to reorganizing the element database so that the lines that will be used in any range are uniformly distributed throughout the database. This would enable each processor to only scan a portion of the database and use the lines that it finds in that section. In addition to reducing the overhead of every processor scanning the entire database, it also simplifies the scheduling problem.

4.2.3. Generation of the shape function

The voigt() function that generates the shape function depends ultimately on three variables: mass (element.lst), temp (spectrap structure of comp.lst), and broaden (broadening parameter from comp.lst). If these three variables either 1) do not change or 2) can be determined ahead of time, the voigt() function can be used to generate a table of shape functions during initialization. This would change the code in the line function so that the shape function is retrieved from a table rather than generated in real-time.

This enhancement would undoubtedly have the biggest effect on the performance

of the software. If this portion of the software is not optimized the performance of the program will be limited by the time taken to generate the shape function(s). Figure 4-21 shows the execution time for the worker part of the code along with the execution time for the serial Version using the same number of elements and lines as the multi-tasking Version. In addition, the time taken for the worker part of the calculations when only one element with one line is being analyzed. This illustrates the best possible performance with the current algorithms. From the information gained from this timing analysis it is apparent that 7 or 8 processors can utilize most available parallelism in SPECTRA.

5. Multidimensional minimization

Although the SPECTRA code can be used to calculate a spectrum given the metal parameters, in the present research it is required that the inverse proble be solved. The spectrum is observed and the metal parameters must be extracted. One way to solve this problem is to consider it as a minimization problem. A single-valued, multiple-variable function is developed which indicates the degree of difference between the observed spectrum and the one calculated by SPECTRA. The input metal parameter values (the function variables) are changed in such a way that the resultig difference is minimized. In the present work, the sum of the absolute values of the differences between the observed and calculated thick spectra is used as the measure of difference to be minimized. This report presents two distinct algorithms that have been used in the process of the present research, the downhill simplex method and CFSQP a package based on Sequential Quadratic Programming.

5.1. Downhill simplex method

This method requires only function evaluations, not derivatives. It is not very efficient in terms of the number of function evaluations that it requires. It is quite robust though, and was used in this research to experiment with multiple calls to the SPECTRA code. This method has a relatively simple geometrical interpretation. A simplex is the geometrical figure consisting, in N dimensions, of N + 1 points (or vertices) and all their interconnecting line segments and polygonal faces, which encloses a finite inner N-dimensional volume. In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron. If any point of a nondegenerate simplex is taken as the origin, then the N other points define vector directions that span the N-dimensional vector space. In multidimensional minimization, the algorithm is given a starting guess, that is an N-vector of independent variables as the first point to try. The algorithm then moves downhill until it encounters a minimum (possibly local). The downhill simplex method must be started not just with a single point, but with N + 1 points, defining an initial simplex. One of these points is used as the initial starting point P_0 and the other N points can be created using linear cobinations of the N unit vectors. The downhill simplex method takes a series of steps, mostly moving the point of the simplex where the function is largest through the opposite face of the simplex to a lower point. In the present work, a function called amoeba() is used to implement the algorithm. A description of this algorithm can be found in the book "Numerical Recipes in C", by W.H. Press et.al., Cambridge University Press.

File spfit.c contains the necessary functions to use amoeba() for either all twenty metals or a smaller set of three metals: titanium, cobalt and nickel. Function spfit() creates the information in all necessary data structures so that spectra may be called repetitively, and initializes the arrays which are passed into the call to the amoeba function. One more argument passed to amoeba() is the address of the function to be minimized called fitfunc(). This function receives a vector of input parameters, which are the concentration and broadening parameter values of the metals, calls spectra(), calculates the sum of the absolute values of the differences between the observed and calculated thick spectra and returns this value.

5.2. CFSQP package

CFSQP is a set of C functions for the minimization of the maximum of a set of smooth objective functions (possibly a single one) subject to general smooth constraints. If the initial guess provided by the user is infeasible for some inequality constraint or some linear equality constraint, CFSQP first generates a feasible point for these constraints; subsequently the successive iterates generated by CFSQP all satisfy these constraints. CSFQP requires functions that define the objective functions and constraint functions and may use used-supplied functions to compute the respective gradients or it can estimate them by forward finite differences.

File sqfit.c contains similar functions as the ones described for the use of amoeba() but it uses function cfsqp() to perfrom the minimization. Although CFSQP supports nonlinear contraint functions, they are not used in the present work. Only linear contraint values are used to identify the proper range of values for eac input variable.

Note: The CFSQP software has been aquired from its authors for use at UAH. Further use outside of UAH requires additional permission from its authors. Details are in the User's Guide for CFSQP Version 2.4, by Craig T. Lawrence, Jian L. Zhou, and Andre L. Tits, University of Maryland. College Park, Md 20742.

APPENDIX A. Makefiles for VxWorks and SUN-UNIX

```
FILE makefile.vx

# compile for VXWORKS
  #DEFINES = -DDOCACHE -DNEW_CONVOLVE -DUSE_LINES -DCROSS -DCHECK -DNOCOLOR -DREPORTS -DDEBUG
  DEFINE_1 = -DDOCACHE -DNEW_CONVOLVE -DUSE_LINES -DCROSS -DCHECK -DNOCOLOR

  DEFINE_2 =  -DCPU=MC68030 -DVXWORKS -DFLOAT=double -DFACTOR=\*10 -DSPECSIZE=50000
  MYCC = cc68k -c $(DEFINE_1) $(DEFINE_2) -I/holo2h/VXWorks/h
  MYLD = ld68k -o
  LDOPTS1 = -r
  LDOPTS2 =

MAIN_FILES = vxmain. main. wav. load. reade. readt. spectra. assign. spectrb. parallel.\
              func. event.
MINOR_FILES = abs. voi. worker. parallel. func. event.

all: main minor

main    :       $(MAIN_FILES:.=.o)
        $(MYLD) main  $(LDOPTS1) $(MAIN_FILES:.=.o) $(LDOPTS2)
        cp main main.VERSION6

minor   :       $(MINOR_FILES:.=.o)
        $(MYLD) minor  $(LDOPTS1) $(MINOR_FILES:.=.o) $(LDOPTS2)
        cp minor minor.VERSION6

vxmain.o:       vxmain.c
        $(MYCC) vxmain.c

main.o  :       main.c spectra.h common.h parallel.h mtest.c nrutil.c amoeba.c spfit.c
        $(MYCC) main.c

assign.o:       assign.c
        $(MYCC) assign.c

wav.o   :       wav.c spectra.h common.h
        $(MYCC) wav.c

load.o  :       load.c spectra.h common.h
        $(MYCC) load.c

reade.o         :       reade.c spectra.h common.h
        $(MYCC) reade.c

readt.o         :       readt.c spectra.h common.h
        $(MYCC) readt.c
```

```
abs.o   :       abs.c minor.h
        $(MYCC) abs.c

func.o  :       func.c
        $(MYCC) func.c

voi.o   :       voi.c minor.h
        $(MYCC) voi.c

spectra.o       :       spectra.c spectra.h common.h parallel.h
        $(MYCC) spectra.c

spectrb.o       :       spectrb.c spectra.h common.h parallel.h
        $(MYCC) spectrb.c

worker.o:       worker.c minor.h common.h parallel.h
        $(MYCC) worker.c

minor.o         :       minor.c minor.h common.h parallel.h
        $(MYCC) minor.c

parallel.o:     parallel.c parallel.h parallel_sun.c parallel_sun.h parallel_vx.c parallel_vx.h
        $(MYCC) parallel.c

event.o         :       event.c event1.c
        $(MYCC) event.c

clean:
        rm *.o
        rm main
        rm minor
```

```
FILE makefile.sun

= compile for Suns
  #DEFINES = -DDOCACHE -DNEW_CONVOLVE -DUSE_LINES -DCROSS -DCHECK -DNOCOLOR -DREPORTS -DDEBUG
  DEFINE_1 = -DDOCACHE -DNEW_CONVOLVE -DCROSS -DCHECK -DNOCOLOR

  DEFINE_2 =  -DCPU=MC68030 -DUNIX -DFLOAT=double -DFACTOR=\*10 -DSPECSIZE=50000
  MYCC = cc -c $(DEFINE_1) $(DEFINE_2)
  MYLD = cc -o
  LDOPTS1 =
  LDOPTS2 = -lm

MAIN_FILES = main. wav. load. reade. readt. spectra. assign. spectrb. parallel. func. event.
MINOR_FILES = abs. voi. worker. parallel. func. event.

all: main minor

main    :       $(MAIN_FILES:.=.o)
        $(MYLD) main  $(LDOPTS1) $(MAIN_FILES:.=.o) $(LDOPTS2)

minor   :       $(MINOR_FILES:.=.o)
        $(MYLD) minor  $(LDOPTS1) $(MINOR_FILES:.=.o) $(LDOPTS2)

main.o  :       main.c spectra.h common.h parallel.h mtest.c nrutil.c amoeba.c spfit.c
        $(MYCC) main.c

wav.o   :       wav.c spectra.h common.h
        $(MYCC) wav.c

load.o  :       load.c spectra.h common.h
        $(MYCC) load.c

reade.o         :       reade.c spectra.h common.h
        $(MYCC) reade.c

readt.o         :       readt.c spectra.h common.h
        $(MYCC) readt.c

abs.o   :       abs.c minor.h
        $(MYCC) abs.c

func.o  :       func.c
        $(MYCC) func.c

voi.o   :       voi.c minor.h
        $(MYCC) voi.c

spectra.o       :       spectra.c spectra.h common.h parallel.h
        $(MYCC) spectra.c
```

```
spectrb.o       :       spectrb.c spectra.h common.h parallel.h
        $(MYCC) spectrb.c

parallel.o:     parallel.c parallel.h parallel_sun.c parallel_sun.h parallel_vx.c parallel_vx.h
        $(MYCC) parallel.c

worker.o:       worker.c minor.h common.h parallel.h
        $(MYCC) worker.c

event.o         :       event.c event1.c event_SGI.c event_vx.c
        $(MYCC) event.c

clean:
        rm *.o
        rm main
        rm minor
```

APPENDIX B. Shell Scripts

The following example is a shell script that is used by VME01 (the coordinator) upon system startup:

SCRIPT STATEMENTS	COMMENTS
loginUserAdd("hecht","cQReycSeRc");	
cd "VXWORKS" ld < vx_init.o vx_init	load and run vx_init to set up the prompt and assign the processor number to the processor
putenv "OPADHOME = host:SPECTRUM/IDL_FILES"	
cd "host:VXWORKS/SCRIPTS/CODE" ld 0,0,"ld_code.o" ld_code	load and run ld_code to read the job description file for this processor download the proper coordinator code

ld 0,0,"run_vme01.o"
cd "host:VXWORKS/SCRIPTS"
sp(run_code,0,0,0,0,0,0,0,0,0);
	load and run run_vme01.o that reads the job description file and calls the coordinator function with the specified command line parameters.

The loginUserAdd() functions allows the specified user to log in with the password specified. If a user is not added to the system in this way, he will not be able to log in to the computer. The next three lines load and run vx_init, a program that sets up the prompt, assigns the processor its processor number and provides the code to perform the reset_all() function that resets all of the processors on the bus. The next line calls the putenv() function which adds the specified environment variable.

The next three lines load and run ld_code, the program that reads the "job description" file for this processor and determines whether to download the code for the coordinator or the worker. The following line downloads and runs run_vme01 that reads the job description file and calls the coordinator function with the specified command line parameters. After the coordinator function returns, the reset_all() function is called. The run_code() function from run_vme01.o is spawned as a separate task so the shell doesn't wait for it to finish.

The shell script for vme02-vme06 is similar to the above except run_vme.o is downloaded and executed instead of run_vme01.o. The main difference is that the worker processors do not reset the system after they finish executing. Figure B-1 shows the major steps of the programs called by the shell scripts.

The following is an example of a "job description" file for the coordinator:

```
5 host:SPECTRUM/work/main.VERSION6 @vxmain -t0 -p5 -n1 -m1 -Rhost:VXWORKS/SCRIPTS/VERSION6/vme01_5.out -f0 VERSION6/
4 host:SPECTRUM/work/main.VERSION6 @vxmain -t0 -p4 -n1 -m1 -Rhost:VXWORKS/SCRIPTS/VERSION6/vme01_4.out -f0 VERSION6/
```

```
3 host:SPECTRUM/work/main.VERSION6 @vxmain -t0 -p3 -n1 -m1 -Rhost:VXWORKS/SCRIPTS/VERSION6/vme01_3.out -f0 VERSION6/
2 host:SPECTRUM/work/main.VERSION6 @vxmain -t0 -p2 -n1 -m1 -Rhost:VXWORKS/SCRIPTS/VERSION6/vme01_2.out -f0 VERSION6/
1 host:SPECTRUM/work/main.VERSION6 @vxmain -t0 -p1 -n1 -m1 -Rhost:VXWORKS/SCRIPTS/VERSION6/vme01_1.out -f0 VERSION6/
```

The fields contain: the number of processors to use, the fully specified filename of the executable for the coordinator, the command line used to invoke the coordinator (including the path for the report file) and the directory where the comp.lst files can be found.
The following is an example of a "job description" file for the workers:

```
0 host:SPECTRUM/work/minor.VERSION6 host:VXWORKS/SCRIPTS/VERSION6/
0 host:SPECTRUM/work/minor.VERSION6 host:VXWORKS/SCRIPTS/VERSION6/
0 host:SPECTRUM/work/minor.VERSION6 host:VXWORKS/SCRIPTS/VERSION6/
0 host:SPECTRUM/work/minor.VERSION6 host:VXWORKS/SCRIPTS/VERSION6/
0 host:SPECTRUM/work/minor.VERSION6 host:VXWORKS/SCRIPTS/VERSION6/
```

The fields contain, the worker's processor number, the fully specified filename for the executable for the worker and the path for the report file.

The function call to start the coordinator is:

```
start(char *cmd);
```

where cmd is a string containing the command line parameters to be passed to the main() function. start() is only used with VxWorks. It merely translates the command string into the main(int argc, char *argv[]) interface used with the UNIX programs and then calls the main function with the correct parameters.

The function call to start the worker processes is:

```
worker(int proc_num, int debug, char *outpath);
```

where proc_num is the processor number assigned to this worker, debug is a value used to select different levels of messages to aid in debugging the code, and outpath is the path that the report file should be created in. The report filename will be created from the processor number and the number of processors participating in the run. For example vme02_3.out is the report file for processor number 2 for the test run in which 3 processors participated in the calculations.

The command line required to start the coordinator in UNIX is:

```
main -m15 -f4 -t0 -p5 -R/home/ebs330/grad/ece/hecht/VXWORKS/SCRIPTS/UNIX/vme01_5.out VERSION6/
```

When the -m# option is chosen, only the path for the "comp.lst" file is specified, not the filename. The mtest fuction creates the filenames for the number of comp.lst files specified (comp00.lst, comp01.lst, comp02.lst etc..).

When the Worker processes are created, the path specified for the coordinator report file is also used for the workers - only the filenames are changed.

vx_init

- Call function gethostname() which extracts a the unique processor name from its ROM.
- The processor name is of the form "vme0X". where currently X=1,...,6.
- Set the processor number to X and the shell prompt equal to the processor name.

ld_code

- Depending on the processor number, determine the job description file name
- Open the job description file
- Read one line to determine the name of the executable SPECTRA file
- Load the executable SPECTRA file into memory
- Close the job description file

run_code

- Depending on the processor number, determine the job description file name
- Open the job description file
- Remove one line from the job description file
- Close the job description file
- Execute the specified program with the specified command line parameters:
- If the processor is the coordinator:
  - Call the coordinator program with the command line
  - When program terminates, reset all processors
- If the processor is a worker:
  - Call the worker program with the command line

Figure 5-1. Summary of programs called by scripts

Name Database: The name database is used to associate a shared memory object's ID with a unique name. The database provides a name-to-value and value-to-name translation. This provides a convenient way to obtain an object's ID (which is the same on all CPUs) from its name.
STATUS smNameAdd(char *name, void *value, int type): This routine adds a *name* of a specified object *type* and *value* to the shared memory objects database. The *name* parameter is a null-terminated string and the *type* parameter specifies the type of shared memory object (binary semaphore, counting semaphore, message queue etc..). The *value* parameter is the object ID or the global address of a block of shared memory allocated with smMemMalloc() or smMemCalloc().
STATUS smNameFind(char *name, void **pValue, int *pType, int waitType): This routine searches the shared memory objects name database for an object matching a specified *name*. If the object is found, its value and type are copied to the addresses pointed to by *pValue* and *pType*. The value of *waitType* can be one of the following:
NO_WAIT (0): The call returns immediately, even if name is not in the database.
WAIT_FOREVER (-1): The call returns only when name is available in the database. If name is not already in, the database is scanned periodically as the routine waits for name to be entered.

Semaphore Functions: The semaphores used in the software are binary semaphores. The routines below handle the creating, taking and giving of the semaphores. VxWorks shared memory object package does not allow shared semaphores to be deleted.
SEM_ID semBSmCreate(int options, SEM_B_STATE initialState): This routine allocates and initializes a shared memory binary semaphore. The semaphore is initialized to an *initialState* of either SEM_FULL or SEM_EMPTY. The *options* parameter specifies the queuing style for the shared memory semaphores (SEM_Q_FIFO). The function returns a semaphore ID used to identify the semaphore.
STATUS semTake(SEM_ID semId, int timeout): This routine performs the take operation on a specified semaphore. A timeout in ticks may be specified by the *timeout* parameter. WAIT_FOREVER and NO_WAIT are additional values that can be used for the *timeout* parameter.
STATUS semGive(SEM_ID semId): The routine performs the give operation on a specified semaphore.

Message Queue Functions: The following functions are used to setup the shared memory message queues. VxWorks shared memory object package does not allow shared message queues to be deleted.
MSG_Q_ID msgQSmCreate(int maxMsgs, int maxMsgLength, int options): This routine creates a shared memory message queue capable of holding up to *maxMsgs* messages, each up to *maxMsgLength* bytes long. It returns a message queue ID used to identify the created message queue.
STATUS msgQSend(MSG_Q_ID msgQId, char *buffer, UINT nBytes, int timeout, int

priority): This routine sends the message in *buffer* of length *nBytes* to the message queue *msgQId*. The *timeout* parameter specifies the number of ticks to wait for free space if the message queue is full. The *timeout* parameter can also have a value of NO_WAIT or WAIT_FOREVER. The *priority* parameter specifies the priority of the message being sent (MSG_PRI_NORMAL, MSG_PRI_URGENT) which determines whether the message is added to the tail of the list or to the head of the list.
int msgQReceive(MSG_Q_ID msgQId, char *buffer, UINT maxNBytes, int timeout): This routine receives a message from the message queue *msgQId*. The received message is copied into the specified *buffer*, which is *maxNBytes* in length. If the message is longer than *maxNBytes*, the remainder of the message is discarded. The *timeout* parameter specifies the number of ticks to wait for a message to be sent to the queue, if no message is available when msgQReceive() is called. The *timeout* parameter can also have a value of NO_WAIT or WAIT_FOREVER.

Shared Memory Functions: These functions allow tasks on different CPUs to allocate and release variable size chunks of memory that are accessible from all CPUs with access to the shared memory system. The shared memory can be mapped into different addresses on different CPUs. Two functions are provided to convert local addresses to global addresses (when adding the object to the name database) and from global addresses to local addresses (when retrieving the object from the name database).
Void *smMemMalloc(unsigned nBytes): This routine allocates a block of memory from the shared memory system partition whose size is equal to or greater than nBytes. The return value is the local address of the allocated shared memory block.
STATUS smMemFree(void * ptr): This routine takes a block of memory previously allocated with smMemMalloc() or smMemCalloc() and returns it to the free shared memory system pool.
Void *smObjLocalToGlobal(void * localAdrs): This routine converts a local shared memory address localAdrs to its corresponding global value.
Void *smObjGlobalToLocal(void *globalAdrs): This routine converts a global shared memory address globalAdrs to its corresponding local value.

APPENDIX D. Parts of common.h

```
typedef struct
{
 int size;
 DB *data[6];       /* pointers to 6 arrays of FLOATS */
 int offset[6];     /* offset from the start of the data array shared mem to the */
}           DATATYPE;

typedef struct
{
 char name[100];
 double conc;
 char in;
 double broadp;
}           ELEMENT;

typedef struct
{
 double temp;
 double pathl;
 double press;
}           SPECTRAP;

typedef struct
{
 int shiftpars;
 double shift[8];
 double resp[3];
}           SPECTRAFIT;

typedef struct
{
 int elem;
 int start_line;
 int num_lines;
}           ELEM_BLK;
```

```
typedef struct
{
 double conc;              /* values from element.lst */
 double mass;
 double sigh2;
 double sigh2o;
 double partf;
 char nist;
 char filn[MAX_ELEM_NAME];
 DATATYPE datn;            /* data from *.bn3 */
 DATATYPE datl;
 double broaden;           /* values from comp.lst */
 double xi;
 ELEMENT els;
}              ELEM_INFO;

typedef struct
{
 ELEM_BLK elem_blk[MAX_ELEM_BLKS];
 int num_elem_blks;
 int total_lines;
}              ASSIGN_ELEM;

typedef struct
{
 int doevent;
 int nothin;
 int reports;
 int check;
 int nocolor;
 int DatnArraySize;
 int repetitions;
 ASSIGN_ELEM assign[MAX_NUM_PROCS];
}              FIXED_GLOBALS;
```

```
typedef struct
{
 SPECTRAP spectrap;
 int linst;
 double llimit;
 double range;
#ifdef CHECK
 double test1[TRANSDIM];
 double test2[TRANSDIM];
#endif
#ifdef NOCOLOR
 double wavarray[TRANSDIM];
 int elem[TRANSDIM];
 double emiss[TRANSDIM];
#endif
}              SHARED_GLOBALS;

extern DB *datn_array;
extern ELEM_INFO *elem_info;
extern SHARED_GLOBALS *Sh_globals;
extern FIXED_GLOBALS *Fx_globals;
extern int elem_cost[NUM_ELEM];
extern ELEM_BLK elem_blk[MAX_ELEM_BLKS];
```

APPENDIX E. Command line options

The available command line options include:

-F# fit (0)
  #=1 don't save fit values
  #=2 save fit values in file.fit
  use -f1 or -f2 to indicate contents of file.thc
-f# save files (1)
  #=1 -> save thick-thin spectrum in X.thc-thn
  #=2 -> save rwave and thick-thin in X.thc-thn
  #=3 -> save rwqave X.thc-thn X.sp1
  #=4 -> save rwqave X.thc-thn X.sp1 nonzero
-e# disable event timing (1)
-m# call mtest function (0)
-d# debug level (0)
-r# report level (2)
-t# nothin (1)
-n# nocolor (0)
-c# check
-C# cross
-p# number of procs to use (1)
-wS S=string of two double numbers
-Rpath path = fully specified output filename
-g# gauss
  #=0 triangular
  #=1 gauss
  #=2 parameterized special
  #=8 response in file 'rasresp.dat'
  #=9 response in file whose name follows option