Tools in Support of Program Partitioning in the UNIX Environment

Miron Livny
Barton P. Miller
Jeff Hollingsworth

March 1992
Tools in Support of Program Partitioning in the UNIX Environment

Miron Livny
Barton P. Miller
Jeff Hollingsworth

March 1992

ABSTRACT

This technical report describes research on computerized analysis tools for partitioning application programs. The goal of this research is to develop algorithms and tools that will enable the user of a C application to decide whether the response time of the application can be improved by offloading part of the application to a second machine. This investigation is being conducted at the University of Wisconsin, Computer Sciences Department by Drs. Miron Livny and Bart Miller. Jeff Hollingsworth is their Research Assistant. This report covers the period June - December 1991.1

1 This report written under contract to Dept. 401 IBM Rochester
Contents

INTRODUCTION ................................................................................. 1

DATA GATHERING ........................................................................ 3
Compiling the Application ......................................................... 3
Executing the Application ......................................................... 5
The Trace Files ................................................................. 6
  Header ...................................................................... 6
  File: calls.out ........................................................... 7
  File: cpu.out .............................................................. 7
  File: file.out .............................................................. 8
  File: memory.out ........................................................ 8
  File: parameters.out ................................................... 8
Collapsing the Data ............................................................. 9
Instrumentation Techniques .................................................. 9
  Files: calls.out and cpu.out ......................................... 9
  File: memory.out ........................................................ 12
  File: file.out .............................................................. 13
  File: parameters.out ................................................... 14

DATA ANALYSIS ...................................................................... 14
Trace Analyzer .................................................................... 14
Partitioning Modeler ......................................................... 15
  System Model ................................................................ 15
  Application Model ........................................................ 16
  CPU Time Model .......................................................... 17
  Optimal Partition ............................................................ 18
  Output of PM ................................................................. 22

Trademarks ........................................................................... 26
INTRODUCTION

In commercial data processing, decision support applications are typically used to analyze the state of an enterprise. They apply sophisticated analysis techniques to large amounts of data in order to provide a quantitative profile of the enterprise. Such applications may be used to profile the productivity of a factory or the success of a promotion campaign. While the analysis part of a decision support application is a compute-intensive task, the application also triggers many I/O operations in order to retrieve the data needed for the analysis. The data about the current state, or the history of the enterprise, is commonly stored in a database and has thus to be accessed via a Database Management System (DBMS). By means of embedded SQL statements or special database access utilities provided by languages like C/400, decision support applications communicate with the DBMS to retrieve data. The data retrieval part of the application is in most cases I/O bounded and is executed on the machine that stores the database. The compute-intensive tasks of the application, however, can be executed by a different machine that is better suited for such tasks. Such a partitioning of the application does not necessarily entail a reduction in response time. The cost of moving data back and forth between the two machines may be higher than the reduction in the execution time of the compute-intensive tasks.

This report presents a toolset that supports the partitioning of C applications in the UNIX environment. For a given application, input data, and pair of machines, our toolset enables the user to tell whether the application can benefit from partitioning, to what extent, and what is the optimal partitioning of the application. The type of partitioning investigated is at the UNIX module (object file) and procedure level (within module).

If a program is to be partitioned into I/O-intensive and compute-intensive parts, there must be a clear boundary on which this partition is established. If control flows too frequently between the partitions (via procedure calls), then this interaction will cause excessive inter-machine communication. If data in one partition is frequently needed in another partition, then the transferring of this data could also cause excessive inter-machine communication.

In order to predict the response time of a given partitioning, we need to know the CPU and I/O consumption, the control flow, and the usage of pointers of each module of the C application, and to translate this information to inter-module interaction for a given partitioning of the program. Existing profiling tools, such as UNIX Gprof, can be used to measure the CPU consumption and control flow of the program. New tools have been developed, however, to record the pointer usage pattern of the application. This section contains an overview of the functionality of the toolset that was developed and a discussion of the interaction of the tools.
The Partitioning Toolset consists of four major parts:

- Extended C Compiler (ECC)
- Extended Libraries (EL)
- Trace Analyzer (TA)
- Partitioning Modeler (PM)

The C applications to be studied are compiled by the ECC. This compiler has several options to select from among the different metrics (described in the next section). The ECC produces static data, including such information as numbers and sizes of procedure parameters, and the extended object code (EOC). The EOC includes instrumentation code to generate traces of dynamic data. The EL is linked in to the EOC to help instrument certain operations. When the EOC executes, it produces dynamic data that includes such information as number of times that a procedure was called and memory references to shared data. The TA processes and summarizes the static and dynamic data generated by the different measurement tools. The PM uses the data from the TA to search for an optimal partitioning.

Section 3 of the report describes the facilities for collecting the data needed for the partitioning algorithm. The steps for compiling and executing the application are described. In addition, the contents of each trace file generated and the mechanisms that generated the files are described. Section 4 presents the analysis portion of the Partitioning Toolset.
DATA GATHERING

This section describes the design and implementation of the facilities for gathering data about a program execution. The application program is compiled with the extended compiler, then executed to produce several trace files. No changes are necessary to the application program.

Compiling the Application

When the application program is compiled (see Figure 1) by the extended C compiler (ECC), two object modules are produced. The first object module (ECO-gprof) contains the standard UNIX gprof instrumentation. The ECO-gprof module is used to collect data about CPU use and parameter sizes that are passed. When this object module is executed, it will produce profile data in a file called "gmon.out." The "gmon.out" file is then processed by the extended gprof utility (described in the next section). The second object module (ECO-fmp) contains instrumentation to collect data about file, memory, and parameter use. This object module is linked with the extended libraries (EL).
Figure 1: Compiling an Application
Executing the Application

First, the two object modules (ECO-gprof and ECO-fmp) are executed. The ECO-gprof object module produces the standard gprof trace output file (called "gmon.out"). The ECO-fmp object module produces three trace files. These trace files contain file reference data ("file.out"), memory reference data ("memory.out") and volume of parameters passed ("parameters.out").

After the object modules have been executed, the extended gprof utility is run, processing the "gmon.out" file to produce two trace files. These trace files contain CPU use data ("cpu.out") and number of procedure calls ("calls.out").

The format and contents of each of the five traces files is described in the next section.
<table>
<thead>
<tr>
<th>File Name</th>
<th>Produced by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>calls.out</td>
<td>ECO-gprof (and EGPROF)</td>
<td>Number of procedures calls made by each pair of procedures</td>
</tr>
<tr>
<td>cpu.out</td>
<td>ECO-gprof (and EGPROF)</td>
<td>Amount of CPU time used by each procedure.</td>
</tr>
<tr>
<td>file.out</td>
<td>ECO-fmp</td>
<td>Amount of I/O (# of reads, # of writes, and number of bytes read and written) done by each procedure, for each file.</td>
</tr>
<tr>
<td>memory.out</td>
<td>ECO-fmp</td>
<td>Amount of memory referenced (# of references and number of bytes) between each pair of procedures.</td>
</tr>
<tr>
<td>parameters.out</td>
<td>ECO-fmp</td>
<td>Amount of parameter passing (# of parameters and number of bytes) between each pair of procedures.</td>
</tr>
</tbody>
</table>

**The Trace Files**

All of the trace files have a similar format. They share a common header format. First, the header format is described and then the entries in each type of trace file.

The information in each type of file is listed by procedure. For example, the number of bytes of I/O is listed for each procedure, rather than each module. Information about the grouping of procedures into modules is also included, so module-based results can easily be derived.

**Header**

The header of each file contains an identification line, followed by the list of procedures. The identification line is in the following format:

```
<version number>  <file type>
```

The `<version number>` describes the version number of the instrumentation software that generated the file. The analysis tools check this number before processing the file. The `<file type>` identifies the type of data contained in this file. Valid values for this file are "calls", "cputime", "io", "memory", and "parameters."
Following the identification line is the list of procedures that were executed by the application. The first line of the list contains the number of entries in the list. Each entry is on its own line and is in the following format:

< name > < procedure number > < module number >

The < name > is the name of a module (file) or procedure. The < procedure number > is a unique integer identifying the procedure. If the entry is for module name, then the procedure number will be -1. The < module number > is a unique integer identifying the module. Each procedure in the same module has the same module number. Modules correspond to UNIX .c source files.

File: calls.out

Each entry in the "calls.out" file describes the number of calls one procedure made to another procedure. Each entry is on its own line and is in the following format:

< procedure #1 > < procedure #2 > < call count >

This entry says that < procedure #1 > called < procedure #2 > the number of times listed in < call count >.

File: cpu.out

Each entry in the "cpu.out" file describes the number of cpu seconds used by a procedure. Each entry is on its own line and is in the following format:

< procedure > < cpu time >

This entry says that < procedure > executed for < cpu time > seconds.

File: file.out

The information in "file.out" file describes the amount of I/O performed by each procedure. Data for each file is described by a sequence of entries. This sequence is repeated for each file that was opened during execution. The first entry in the sequence has the following format:

< file name > < procedure > < numprocs >

This entry says that file < file name > was opened by < procedure >.
This entry is followed by `<numprocs>` other entries, one entry for each procedure that read or wrote the file. These entries have the following format:

```
<procedure> <reads> <read bytes> <writes> <write bytes>
```

This entry says that `<procedure>` made read and write accesses to the file described by the above entry. There were `<reads>` read accesses, totaling `<read bytes>` bytes, and `<writes>` write accesses, totaling `<write bytes>` bytes.

**File: memory.out**

Each entry in the "memory.out" file describes the number of cross-module references made by a procedure. Each entry is on its own line and is in the following format:

```
<procedure #1> <procedure #2> <reads> <read bytes> <writes> <write bytes>
```

This entry says that `<procedure #2>` made memory read and write accesses to data allocated by `<procedure #1>`. There were `<reads>` read accesses, totaling `<read bytes>` bytes, and `<writes>` write accesses, totaling `<write bytes>` bytes.

**File: parameters.out**

Each entry in the "parameters.out" file describes the amount of parameter passing during the program execution. Each entry is on its own line and is in the following format:

```
<procedure #1> <procedure #2> <bytes>
```

This entry says that `<procedure #1>` called `<procedure #2>`, passing a total of `<bytes>` bytes. The number of times that each procedure called another procedure is found in the file "calls.out."

**Collapsing the Data**

If the application program has a large number of procedures, it is not possible to store per-procedure information for the memory reference data (see the next section for a description of the data structures involved). Therefore, when the number of procedures exceeds a threshold value (the current default is 350), the memory reference data is collected per-module rather than per-procedure. This threshold value can be changed by setting the UNIX environment variable `ECC_LARGE_PROC_COUNT` to the desired value.
The collapsing of data has little effect on the subsequent analyses.

Instrumentation Techniques

This section describes the mechanisms used to gather the data described in the previous sections. Three major techniques are used. First, the GNU C compiler ("gcc") was modified to insert tracing instructions into the application programs. This technique is used to gather parameter usage information and part of the memory access information. Second, the C library was modified to trace relevant events. This technique was used to gather additional memory access information and file access information. Last, we modified the UNIX "gprof" utility to gather the CPU usage and procedure call information.

For each of the trace file types, we describe the methods used to gather the data contained in that file.

Files: calls.out and cpu.out

The UNIX "gprof" utility was modified to produce the calls.out and cpu.out files. First, an application program is compiled to collect gprof profiling information. This information is gathered by inserting profiling code in the application program. When the program is executed, the profiling code uses the "profil" system call to tell the kernel to activate monitoring. The kernel periodically samples the program counter (PC) and increments the table entry corresponding to that PC value by 10 milliseconds. At completion of the application program, this table is stored in a file called "mon.out." The gprof utility reads "mon.out," correlating the accumulated times with procedure names (the range of PC values for a procedure is found, and then the table entries for this range is summed).

We modified the gprof utility to generate a matrix describing inter-procedure calls. This information is written to the "calls.out" file, described above. A second output file is written that records the amount of CPU time spent in each procedure; this is the cpu.out file.

The main module of gprof ("gprof.c") and the output module ("printgprof.c") were modified to include calls to new instrumentation code. The instrumentation code is contained in a new file named "matrix.c."

File: memory.out

All memory in the application program is defined to be owned by the module or procedure that created it. Global variables are owned by the module in which they are declared. Heap variables are owned by the procedure in which they are allocated. The GNU C compiler was modified to produce calls to a runtime instrumentation library for each cross-module memory reference. It was also modified to record which
modules own global variables. In addition, the standard UNIX heap manager (malloc) was modified to record which procedure allocates each heap variable. Each time a cross-module memory reference is detected, the runtime library computes the owner and user of the memory location, and updates an internal matrix of memory references. In addition, C library routines that operate on memory (e.g. strcpy, bcopy) were modified to record their memory references. This approach was better than simply recompiling the libraries with the modified C compiler because we can record block transfers as a single multi-word transfer rather than a sequence of single word transfers.

![Data Structure to track Heap Ownership](image)

Figure 3: Data Structure to track Heap Ownership

Tracking memory ownership efficiently requires the runtime library to maintain a sophisticated data structure. The data structure used to track the heap is shown in Figure 3. The approach exploits the design of the Berkeley Unix memory allocator. All memory requests are rounded up to the next bucket size. Bucket sizes are powers of two, so a request of 45 bytes would be rounded to 64 bytes. Each page of memory contains heap space for only one bucket size.
Each allocated region of heap memory contains a header for the variable in that region. An additional field was added to this header to record the identity of the procedure that allocated the space. To map an arbitrary memory reference in the heap back to the procedure that allocated it, a data structure was developed that is modeled after a multi-level page table scheme used to implement virtual memory. The indirection table divides the entire 32 bit address space of the application process into 64 sub-regions. Each of the 64 sub-regions contains a page table with 4096 entries. Each of the page table entries correspond to a single 8192 byte page. The contents of the page table depends on the size of the buckets contained in that page. If the bucket size is less than one page, the page table entry is the number of bits to shift an address to find the owner (see the top Data Page in Figure 3). If the page's bucket size is larger than a page, the entry in the page table is the pointer to the structure that describes the procedure (see the bottom data page in Figure 3).

The owner of global memory is tracked in a similar way. However, since the granularity of alignment of global variables is eight bytes, a third level of indirection is used. The page table entry for a global page points to a table that indicates the owner of each eight-byte chunk of memory. The overhead of the approach (in memory use) is fairly high (512 bytes for each 8192 byte page).

The user of a memory location is tracked by keeping a linked list of active procedures. Each node in the list is stored in the activation record for the corresponding procedure activation. When a procedure is called, a new entry in the linked list is created. The new activation is added to the head of the list, and the list head pointer (called "ECCCurrentFunc") is updated to point to this structure (see Figure 4).
07.1770

![Diagram of linked list of active procedures](image)

**File: file.out**

The file I/O instrumentation was built by modifying and extending existing runtime library routines. The instrumentation was constructed so that buffered I/O (e.g., print) would be charged to the procedure that requested the I/O, not the procedure that ultimately flushed the buffer to disk. Data about file I/O is recorded in a two dimensional matrix indexed by file descriptor and by procedure. All of the buffered and non-buffered I/O library routines were instrumented to record the number of bytes transferred and the number of transfers. Instrumentation was added to the open and close system calls. The open system call was modified to record the name of the file being opened, and which procedure was opening it. The close system call was modified to write the statistics about the file being closed to disk. This information includes: the file name, the name of the procedure that opened the file, and the accumulated I/O statistics for that file. This information must be written during the close operation because the statistics are indexed by file descriptor and UNIX reuses closed file descriptors. One final change was required to the dup system call to copy the file information in memory (but not the accumulated statistics) to the newly created file descriptor.
File: parameters.out

Parameter information is collected by modifying the C compiler's procedure call code generation routine. This routine now generates an expression to compute the total size of the parameters being passed to a procedure. The called procedure then records the parameter size information in a two dimensional matrix indexed by the caller and callee. This approach was required because the size of the parameters passed to a procedure can't be determined at compile time (e.g., structures containing dynamic arrays can be passed by value to other procedures).
DATA ANALYSIS

The analysis part of the Partitioning Toolset consists of two elements: the Trace Analyzer (TA) and the Partitioning Modular (PM). The role of these two elements is to summarize the data that was collected by the various measurement elements and to find a partitioning with a minimal CPU time for the measured application. In this section, the functionality of the TA is described and the algorithm employed by the PM to find the optimal allocation is presented.

Trace Analyzer

For each of the application procedures, the TA first collects all the recorded information from the *.out files described in the previous section. It then summarizes this information for each of the compilation units2 of the application. Two files, "UnitReport" and "ProcReport," are generated by the TA (see Figure 5). In "UnitReport" the TA presents a summary of the measured information at the compilation unit level. For each unit \( u \), a summary entry is presented that includes the CPU time, number of I/Os initiated by \( u \), number of inter-unit procedure calls performed by \( u \), and the number of times a procedure in \( u \) was called by a procedure in another unit. Each summary entry is on its own line and is in the following format:

\[
\text{UNIT <unit id #> (<CPU time> <IO count> <out call count> <in call count>)}
\]

If procedures in \( u \) did call procedures in other units or did access memory allocated by another unit, then the entry of unit \( u \) includes a list of these units. Each element in the list includes the identification number of the unit, a count of the number of times the unit was called, and a flag (M) that indicates whether \( u \) accessed memory that was allocated by this unit.

2 (*): From this point on, the term "compilation unit" is used to refer to the modules or *.c files of the application.
The format of the entries in "ProcReport" is the same as the format of the entries in "UnitReport." The only difference between the two files is that the former presents the information at the procedure level whereas the latter presents it at the compilation unit level. In addition to generating the report files, TA passes the summary information to the Partitioning Modeler via shared memory.

**Partitioning Modeler**

The Partitioning Modeler (PM) is the "brain" of the toolset. It takes all the information measured by the various tools, identifies an optimal partitioning for the given application and the given input data and predicts its total CPU time. In addition to the optimal partitioning, the PM provides information on the performance of the application under a number of sub-optimal partitionings. Although the CPU time of this partitioning is higher than the optimal partitioning, the user may prefer these partitionings since they require fewer changes to the original application. The data provided by the PM allows the user to evaluate the cost effectiveness of a range of partitionings. The performance predictions of the PM are based on a system model, an application model, and a CPU cost model; these models are described in the first three
subsections of this section. Based on these models, an algorithm was developed that uses the information collected by the measurement tools to identify an optimal partitioning of the application for the given input data. In the fourth subsection, this algorithm is motivated and presented. The output of the PM is discussed in the last subsection.

System Model

The system consists of two computers, $P_a$ and $P_b$, that are interconnected by a communication subnet. $P_b$ has a CPU that is CPU-speedup times faster than the CPU of $P_a$. The system supports procedure calls across computer boundaries. A procedure on one computer can invoke a procedure on the other computer via a remote procedure call (RPC). Such a remote call adds remoteCall seconds to the execution time of the application. Computer $P_a$ provides the entire suite of I/O operations, whereas $P_b$ can execute only a subset of these operations. When a procedure initiates an I/O operation on $P_b$ that can not be executed locally, the operation is shipped to $P_a$ and is executed there. Once the operation has been executed, the results are transferred back to $P_b$. It takes a total of remoteI/O seconds to ship an I/O operation to $P_a$ and to transfer the results back to $P_b$.

Application Model

An application is composed of PR procedures that reside in $U$ compilation units. For each procedure $i$, the total processing time (in seconds) of the procedure on $P_a$, and the number of I/O operations initiated by the procedure, are $pproc_i$ and $pio_i$ respectively. The interaction between procedures is captured by the procedure call matrix $P_{CALL} (PR,PR)$ and the procedure memory matrix $P_{MEM} (PR,PR)$. The value of $p\text{call}(i,j)$ is the number of times procedure $i$ called procedure $j$. $P_{MEM}(i,j)$ is set to true if procedure $i$ accessed a memory location that was allocated by procedure $j$.

From the execution profile of the procedures, the execution profile of the compilation units can easily be derived. For each compilation unit $c$, we derive the total processing time (in seconds) on $P_a$

$$c_{proc} = \sum_{i \in c} p_{proc_i}$$
and the number of I/O operations

\[ cio_c = \sum_{i \in c} pio_i \]

The matrices \( CCALL(U,U) \) and \( CMEM(U,U) \) are derived in a similar way

\[ ccall(i,j) = \sum_{k \in i} \sum_{l \in j} pcall(k,l) \]

and

\[ cmem(i,j) = \bigcup_{k \in i} \bigcup_{l \in j} pmem(k,l) \]

**CPU Time Model**

The partitioning of the application places each of the application procedures on one of the two system processors. There are no restrictions on where a given procedure can be placed. It is assumed that the application provides sufficient information to the runtime environment so that the latter can support all the inter-procedure interactions and the IO operations regardless of the location of the procedures. Procedures on different machines can interact via remote procedure calls and shared memory.

Let \( loc_i \) be the location of procedure \( i \) under partition \( k \), then the total CPU time of the application under partition \( k \) is

\[ acpu_k = aproc^k_a + aproc^k_b + rio_k + rcall_k \]

where

\[ aproc^k_a = \sum_{loc_i^{k} \in a} pproc_i \]
and

\[ aproc_k = \sum_{loc_i = b} pproc_i \times CPU\ speedup \]

are the processing time of the application on \( P_a \) and \( P_b \) respectively. The last two terms in the definition of \( acpu_k \)

\[ roi_k = \sum_{loc_i = b} pio_i \times bufferHit \times remoteIO \]

and

\[ rcall_k = \sum_{loc_i \neq loc_j} pcall(i,j) \times remoteCALL \]

are the CPU overheads of the remote I/Os and the remote procedure calls respectively.

**Optimal Partition**

The problem of finding the optimal partitioning of an application (i.e. a partition with the minimal \( acpu_k \)) given the above system and application models can be mapped to the problem of finding the maximum flow from a source vertex, \( s \), to a destination vertex, \( d \), in a directed network. The algorithm that finds the maximum flow between two selected vertices in the network will also derive the optimal partition for the given application. A directed network is a digraph (directed graph) in which each edge has a non-negative capacity assigned to it (see Figure 6). The maximum flow from \( s \) to \( d \) is equal to the capacity of the minimal \((s,d)\) cut. An \((i,j)\) cut in a network is a set of arcs, that if removed from the network, there will be no path from vertex \( i \) to vertex \( j \) (i.e. the maximum flow from \( i \) to \( j \) will be zero). In Figure 6, the set of arcs \( C_1 = (a,1),(a,2),(3,2),(3,b) \) is an \((a,b)\) cut. \( C_1 \) is not, however, a minimal \((a,b)\) cut. A minimal \((i,j)\) cut is an \((i,j)\) cut with the smallest total capacity. The capacity of of \( C_1 \) is 1760 whereas the capacity of the \((a,b)\) cut \( C_2 = (a,2),(a,3),(1,b),(1,2) \) is only 220. \( C_2 \) is a minimal cut since there is no other \((a,b)\) cut with a capacity that is smaller than 220. An \((s,d)\) cut partitions the network into two sets of vertices:
Those that can be reached from $s$ (1 in the case of $C_2$) and those that cannot be reached from the source vertex (2,3,b in the case of $C_3$). As we will see later, these two sets define the optimal partitioning of the application.

The mapping of the partitioning problem to the maximum flow problem goes as follows: Draw a directed network with vertices the $PR$ procedures 1,2,...,$PR$ of the application. Add a source vertex, $a$, and a destination vertex, $b$, and include arcs from $a$ to all procedures and from all procedures to $b$. For each pair of procedures $i$ and $j$ that meet the condition $pcall(i,j) + pcall(j,i) > 0$ include an arc$(i,j)$ and an arc$(j,i)$. $c_{ij} = pcall(i,j) + pcall(j,i)$. On arc $(i,j), i, j = 1,2,...,PR$, place a capacity

$$c_{ij} = (pcall(i,j) + pcall(j,i)) \times \text{remoteCALL}$$

On arc $(a,j), j = 1,2,...,PR$, place a capacity

$$c_{a,j} = \max(0.0, \frac{pproc_j}{CPU\text{speedup}} + pio_j \times \text{hitRatio} \times \text{remoteIO} - pproc_j)$$
**Table 1. System Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU speedup</td>
<td>10.0</td>
</tr>
<tr>
<td>remoteCall</td>
<td>10.0</td>
</tr>
<tr>
<td>remoteIO</td>
<td>10.0</td>
</tr>
<tr>
<td>hitRatio</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Table 2. CPU and IO information**

<table>
<thead>
<tr>
<th>proc</th>
<th>$cpu_i$</th>
<th>$io_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>500</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 3. PCALL (PR,PR)**

<table>
<thead>
<tr>
<th>proc</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>
and on arc \((i,b), i = 1, 2, ..., PR\), place a capacity

\[c_{ib} = \max(0.0, pproc_i - \left(\frac{pproc_i}{CPU\text{ speedup}} + pio_i \times hitRate \times remoteIO\right))\]

The resulting network captures both the system model and the application model. The capacity of \((a,i)\) and \((i,b)\) tells us the loss in CPU time if procedure \(i\) is executed on \(P_b\) or \(P_s\) respectively. For each pair of procedures \(i\) and \(j\), the capacity of \((i,j)\) tells us the CPU overhead that will be incurred if the two procedures are not assigned to the same processor. In Tables 1-3 we present the system and application parameters that were used to generate the network in Figure 6.

The number of steps required to find the maximum flow from \(a\) to \(b\) is less than \([n(n-1)]n\) where \(n = PR + 2\) is the number of vertices in the network. As a byproduct, our maximum flow algorithm which uses flow-augmenting chains also identifies a minimal capacity \((a,b)\) cut, \(C\). This cut separates the set of procedures into two sets: the procedures that can be reached from \(P_s\), \(A\), and the procedures that can not be reached from \(P_s\), \(B\). Given the way capacities were assigned to the arcs of the network, the total capacity of
C is the total loss in CPU time due to the placement of the set A on \(P_a\) and the set B on \(P_b\), and the cost of remote procedure calls between procedures in A and in B. Therefore, since C is the minimal cut, a partition, \(k\), that assigns the procedures in \(A\) to \(P_a\) and the procedures in \(B\) to \(P_b\), has the optimal (smallest) total CPU time \(a_{cpu}\). Since \(C_2\) is the minimal \((a,b)\) cut, the optimal partitioning of the application that is represented by the network in Figure 6 is procedure 1 on \(P_a\) and the other two procedures on \(P_b\).

An intuitive explanation for why the partition generated by \(C\) is optimal goes as follows: for each procedure \(i\) the \((a,b)\) cut, \(C_i\), includes either the arc \((a,i)\) or the arc \((i,b)\). Since \(C\) is an \((a,b)\) cut, it has to include at least one of these two arcs and since it is a minimal cut, it cannot include both. It is also clear that for every pair of vertices \(i\) in \(A\) and \(j\) in \(B\), the arc \((i,j)\) has to be in \(C\) if the arc is in the original network. Therefore, if \(i\) is placed by \(C\) on \(P_a\) \((i\in A)\), we know that the cost of placing it on \(P_a\) (the capacity of \((a,i)\)), plus the cost of the remote calls it performs for procedures on \(P_b\) and procedures on \(P_b\) call \(i\) \((\sum_{j\in B} \text{capacity}(i,j))\), is smaller than the cost of placing it on \(P_b\) (the capacity of \((i,b)\)) plus the overhead of its remote interaction with procedures on \(P_a\) \((\sum_{j\in A} \text{capacity}(i,j))\).

If this was not true, we could have placed \(i\) on \(P_b\) to obtain an \((a,b)\) cut with a smaller capacity than \(C\), which contradicts the minimality of \(C\).

Output of PM

Table 4 presents an example of the report the Partition Modeler produces on the screen. In addition to this report it generates two placement files, UnitPlacement and ProcPlacement. The first and second section of the report present information on the system and the application respectively. Most of the entries in these two sections are self-explanatory. The entry in the application part that presents the CPU time of the application provides two values, one for \(P_a\) and one for \(P_b\). Both values are pure CPU times and do not include any CPU times for remote I/O or RPCs.

The third section of the PM report presents the predicted CPU time of the application under the optimal partition. In addition to the CPU information, a count of procedures allocated to each processor, the number of RPCs, and the number of remote I/Os for the optimal partitioning is also provided. A listing of the partitioning (i.e. which procedures are placed on \(P_a\) and which on \(P_b\)) is included in ProcPlacement. The second CPU time in this entry is the predicted CPU time of the application under an optimal assignment that meets the requirements that all the procedures that belong to the same compilation unit are placed on the same processor. In other words, this is the optimal partitioning at the compilation unit level. The difference between the first and second CPU times gives us an indication as to how much can be gained by breaking the compilation units into smaller units. As in the case of the optimal procedure partitioning, the PM also presents for the optimal unit assignment the number of units allocated to each processor, the
number of remote I/Os, and the number of RPCs. A listing of the partitioning (i.e. which units are placed on $P_a$ and which on $P_b$) is included in UnitPlacement.

Under the title AssOptNoM, the PM presents the CPU times of the optimal procedure and unit partitionings that meet the following constraint: a procedure/unit that accesses memory that was allocated by another procedure/unit has to be placed on the same processor as the procedure/unit that allocated the memory. These are assignments that do not require the support of a mechanism that allows sharing of memory across processor boundaries. The difference between the CPU time reported in this section and the previous section of the report is an indicator of how much we lose if we do not use such a mechanism.
WELCOME TO THE PLACEMENT TOOLS(v1.0)

SYSTEM PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Speed Ratio</td>
<td>1/20 A/B</td>
</tr>
<tr>
<td>Remote Procedure Call</td>
<td>.0009</td>
</tr>
<tr>
<td>Remote IO Overhead</td>
<td>.0009</td>
</tr>
<tr>
<td>IO Buffer Hit Ratio</td>
<td>.4000</td>
</tr>
</tbody>
</table>

APPLICATION INFO

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of modules</td>
<td>43</td>
</tr>
<tr>
<td>Number of procedures</td>
<td>1274</td>
</tr>
<tr>
<td>Total CPU Time</td>
<td>1.74e1 (a) 0.8720 (b)</td>
</tr>
<tr>
<td>Total number of IOs</td>
<td>26994</td>
</tr>
</tbody>
</table>

AssignOpt

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU Time</td>
<td>9.7636</td>
</tr>
<tr>
<td>Procedures per proc</td>
<td>26 (a) 1248 (b)</td>
</tr>
<tr>
<td>Remote Procedure Call</td>
<td>8407</td>
</tr>
<tr>
<td>Remote IOs</td>
<td>649</td>
</tr>
<tr>
<td>Total CPU Time</td>
<td>1.06e1</td>
</tr>
<tr>
<td>Modules per processor</td>
<td>43 (b)</td>
</tr>
<tr>
<td>Remote Procedure Call</td>
<td>0</td>
</tr>
<tr>
<td>Remote IOs</td>
<td>10798</td>
</tr>
</tbody>
</table>

AssignOptNoM

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU Time</td>
<td>9.7636</td>
</tr>
<tr>
<td>Procedures per proc</td>
<td>26 (a) 1248 (b)</td>
</tr>
<tr>
<td>Remote Procedure Call</td>
<td>8407</td>
</tr>
<tr>
<td>Remote IOs</td>
<td>649</td>
</tr>
<tr>
<td>Total CPU Time</td>
<td>1.06e1</td>
</tr>
<tr>
<td>Modules per processor</td>
<td>43 (b)</td>
</tr>
<tr>
<td>Remote Procedure Call</td>
<td>0</td>
</tr>
<tr>
<td>Remote IOs</td>
<td>10798</td>
</tr>
</tbody>
</table>

AssignNaive

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU Time</td>
<td>8.17e1</td>
</tr>
<tr>
<td>Procedures per proc</td>
<td>14 (a) 1260 (b)</td>
</tr>
<tr>
<td>Remote Procedure Call</td>
<td>89153</td>
</tr>
<tr>
<td>Remote IOs</td>
<td>7</td>
</tr>
<tr>
<td>Total CPU Time</td>
<td>5.40e1</td>
</tr>
<tr>
<td>Modules per processor</td>
<td>4 (a) 39 (b)</td>
</tr>
<tr>
<td>Remote Procedure Call</td>
<td>57651</td>
</tr>
<tr>
<td>Remote IOs</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4 Sample Screen Report

The last section of the report presents the behavior of the application under what we call the "naive" partitioning. Here the cost of Remote Procedure Calls is ignored in the partitioning process. Procedures/units are assigned to the processor that will executed the procedure in minimal CPU time.

24
(including remote I/O time). Using the notation of the network presented in 4.2.4, procedure \( i \) will be placed on \( P_a \) if the capacity of \( (a,i) \) is larger than the capacity of \( (i,b) \). Otherwise, it will be placed on \( P_b \). Here the partitioning is based on a local view of each procedure/unit and does not consider information on inter-procedure/unit behavior. We have included this naive approach to demonstrate the power of the global (and more sophisticated) approach to the partitioning problem that the Partitioning Toolset employs. In many cases the predicted CPU time of the naive partitioning is much higher than the CPU time predicted for the application when placed as one unit on either one of the processors.
Trademarks

The following are Trademarks of their respective entities:

- UNIX is a Trademark of UNIX Systems Laboratories Inc.
- C/400 and AS/400 are Trademarks of International Business Machines Corp.
- GNU compiler Software copyrighted and licensed from the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139