SYNERGY V3.0 Usage Manual

Yuan Shi
shi@temple.edu
http://spartan.cis.temple.edu/shi
(215) 204-6437 (Voice) ♦ (215) 204-5082 (Fax)
(c)Temple University, Philadelphia, PA 19122

January 1995
(revised March 2004)
(revised April 2004)

Table of Contents

1 Preface

As uni-processors speeds racing to the 4 billion instructions per second (GIPS) range, it has become increasingly difficult to justify using a parallel processor for an application. Amongst all unfavorable reasons, programming difficulty and lower availability are the most prevailing.

Strange as it may seem, parallel processing of any application is not really a functional requirement. Therefore, we focus only on performance, load balancing and availability issues.

The material disclosed in this document is part of two patents (U.S.#5,381,534, #5,517,656). Parallel processing researchers, faculty members and graduate students are welcome to a free evaluation copy of Synergy. Commercial use of this system or the disclosed information without the written consent by the Technology Transfer Office of Temple University may constitute an infringement to the approved patents.

1.1) What is Synergy?

Synergy is a parallel computing system using a Stateless Parallel Processing (SPP) principle. It is a simplified prototype implementation of a Stateless Machine (SLM). It lacks backbone fault tolerance and stateful process fault tolerance. It is also known to have an inefficient tuple matching engine in comparison to the full implementation of SLM.

SPP is based on coarse-grain dataflow processing. A full SLM implementation will offer, in addition to all benefits that Synergy affords, a more efficient tuple matching engine and a non-stop computing platform with total fault tolerance for stateful processes and for the backbone. An SLM can be considered a higher form of Symmetric MultiProcessor (SMP).

Functionally, Synergy can be thought of as an equivalent to PVM, Linda or MPI/MPICH.

Synergy uses passive objects for inter-process(or) communication. It offers programming ease, load balancing and fault tolerance benefits. The application-programming interface (API) is a small set of operators defined on the supported object types, such as tuple space, file and database. Synergy programs use a conventional open-manipulate-close sequence for each passive object. Each Synergy program is individually compiled using a conventional compiler and a Synergy Language Injection Library (LIL). A parallel application is synthesized through a configuration specification (CSL) and an automatic processor-binding algorithm. Synergy runtime system can execute multiple parallel applications on the same cluster at the same time.

Synergy API blends well into the conventional sequential programs. It is particularly helpful for re-engineering legacy applications. It even allows parallel processing of mixed PVM and MPI programs.

1.2) Why Synergy?

First, one hidden fact that has not been mentioned in any high performance multiprocessor's literature is that the use of multiple processors for a single application necessarily reduces its availability if any processor failure can halt the entire application. The current state of art in parallel processing is still under the shadow of this gloomy fact. SPP offers an approach that promises breakthroughs in both high performance and high availability using multi-processors. Synergy is the first prototype designed to explore architectural flaws and to validate the claims of SPP.

Second, technically, separation of functional programming from process coordination and resource management functions can ease parallel programming while maintaining high performance and availability. Although many believe that explicit manipulation of processes and data objects can produce highly optimized parallel codes, we believe ease of programming, high performance and high availability are of a higher importance in making industrial strength parallel applications using multiprocessors.

1.3) What are in Synergy?

The first important ingredient in Synergy is the confinement of inter-program communication and synchronization (IPC) mechanisms. They convert dynamic application dataflows to a static, bipartite IPC graph. In Synergy, this graph is used to automate process coordination and resource management. In other words, Synergy V3.0 uses this static IPC graph to automatically map parallel programs onto set of networked computers that forms a virtual multiprocessor. In the full SLM implementation, this static IPC graph will be implemented via a self-healing backbone.

Synergy v3.0 contains the following service components:

a) A language injection library (LIL). This is the API programmers use to compose parallel programs. It contains operators defined on supported passive objects, such as tuple space, file, pipe or database.

b) Two memory resident service daemons (PMD and CID). These daemons resolve network references and are responsible for remote process/object execution and management.

c) Two dynamic object daemons (TSH and FAH). These daemons are launched before every parallel application begins and are removed after the application terminates. They implement the defined semantics of LIL operators.

d) A customized Distributed Application Controller (DAC). This program actually synthesizes a multiprocessor application. It conducts processor binding and records relevant information about all processes involved in the application until completion. DAC represents a customized virtual multiprocessor for each application.

e) Synergy shell: (prun and pcheck). These programs are Synergy runtime user interface. Prun launches a parallel application. Pcheck is a runtime monitor for managing multiple parallel applications and processes.

Optimal processor assignment is theoretically complex. Synergy's automatic processor binding algorithm is extremely simple: unless specifically designated, it binds all tuple space objects, one master and one worker to a single processor. Other processors run the worker-type (with repeatable logic) processes. Since network is the bottleneck, this binding algorithm minimizes network traffic thus promising good performance for most applications using the current tuple matching engine. The full implementation of SLM will have a distributed tuple matching engine that promises to fulfill a wider range of performance requirements.

Fault tolerance is a natural benefit of the SPP design. Processor failures discovered before a run are automatically isolated. Worker processor failures during a parallel execution is treated in V3.0 by a "tuple shadowing" technique. Synergy V3.0 can automatically recover the lost data from a lost worker with little overhead. This feature brings the availability of a multiprocessor application to be equal to that of a single processor and is completely transparent to application programs.

Synergy provides the basis for automatic load balancing. However, optimal load balancing requires adjusting tuple sizes. Tuple size adjustments can adapt guided self-scheduling [1], factoring [2] or fixed chunching using the theory of optimal granule size for load balancing [3].

Synergy V3.0 runs on clusters of workstations. This evaluation copy allows unlimited processors across multiple file systems (*requires one binary installation per file system).

1.4) History

Synergy V3.0 is an enhancement to Synergy V2.0 (released in early 1994). Earlier versions of the same system appeared in the literature under the names of MT (1989), ZEUS (1986), Configurator (1982) and Synergy V1.0 (1992) respectively.

1.5) Comparisons with Other Systems

1.5.1) Synergy vs. PVM/MPI

PVM/MPI is a direct message passing system [5,6] that requires inter-process communication be carried out based on process task id's. This requirement forces an extra user-programming layer if fault tolerance and load balancing are desired. This is because for load balancing and fault tolerance, working data cannot be "hard wired" to specific processors. An "anonymous" data item can only be supplied using an additional data management layer providing a tuple space-like interface. In this sense, we consider PVM/MPI a lower level parallel API as compared to Linda and Synergy.

Fault tolerant and load balanced parallel programs typically require more inter-process communication than direct message passing since they refresh their states frequently in order to expose more “stateless moments” – critical to load balance and fault tolerance. This is a tradeoff that users must make before adapting the Synergy parallel programming platform.

1.5.2) Synergy vs. Linda

The original Linda implementation [4] uses a virtual global tuple space implemented using a compile time analysis method. The main advantage of the Linda method is the potential to reduce communication overhead. It was believed that many tuple access patterns could be un-raveled into single lines of communication. Thus the compiler can build the machine dependent codes directly without going through an intermediate runtime daemon that would potentially double the communication latency of each tuple transmission. However, experiments indicate that majority applications do not have static tuple access patterns that a compiler can easily discern. As a result, increased communication overhead is inevitable.

The compile time tuple binding method is also detrimental to fault tolerance and load balancing.

Another problem in the Linda design is the limited scalability. Composing all parallel programs in one file and compiled by a single compiler makes programming unnecessarily complex and is impractical to large-scale applications. It also presents difficulties for mixed language processing.

In comparison, Synergy uses dynamic tuple binding at the expense of increased communication overhead by using dynamic tuple space daemons. In the full SLM implementation, this overhead will be reduced by a distributed tuple matching engine. Practical computational experiments indicate that synchronization overhead (due to load imbalance) logged more time than communication. Thus Synergy's load balancing advantage can be used to offset its increased communication overhead.

2 Personalized Configuration

The use of csh/tcsh is assumed in the following discussion.

2.1) Environment setup

Edit your .cshrc (or the proper equivalent if you do not use csh) add a line:

setenv SNG_PATH ***synergy_directory***

where ***synergy_directory*** is the directory containing all the binary and the Synergy object library.

Then add the synergy binary directory into your path definition:

```
        set path=($SNG_PATH/bin $path)
```

It is important to add $SNG_PATH/bin before $path, since "prun" may be overloaded in some operating systems (such as SunOS 5.9).

To activate the settings, enter:

```
        %source .cshrc
```

You will need to do all above steps once for each cluster, if you intend to use hosts on multiple clusters.

2.2) How to create a processor pool

Setup a local processor pool file is necessary on a host if you will submit parallel jobs from this host. The commands for configuring the local host file are as follows:

shosts - Syntax: %shost <default | ipaddr1 ipaddr2 ... >

Creates a host file (~/.sng_hosts) using /etc/hosts.

Example: %shosts default

This creates a host list using contents in /etc/hosts. If it creates only a single host entry, try this on a NFS client. Some NFS servers maybe configured differently than their clients.

Note1: This program assumes that you have the same login on all hosts. You will have to edit the host file if you have different logins on some hosts.

Note2: The default setting for all host entries is “selected” meaning they are ready for parallel processing. Before each parallel run, you should “select” a subset of hosts for that application. You can use “chosts” or simply edit ~/.sng_hosts by putting “#” sign in front of a host (de-selected).

addhost - Syntax: %addhost <hostname> [-f]

This command adds a host into the host file. The command fails if the given host is not Synergy capable. The [-f] option forces the insertion even if the host is not ready.

A newly added host automatically becomes “selected”.

delhost - Syntax: %delhost <hostname> [-f]

This command permanently deletes a host from the host file. It fails if the host is Synergy ready. The [-f] option forces the removal.

dhosts - Syntax: %dhosts [-v]
This command lets you permanently delete more than one host at a time. The -v option will verify the hosts' current Synergy connection status (it takes some extra time).

sfs - Syntax: %sfs
This command sets file systems for the hosts in the Synergy host file. You should use different symbols to represent different file systems. Hosts in the same cluster should have the same file system symbol.

Note: This version is NOT compatible with earlier versions. Therefore you must RECREATE the local host file and then set file systems using sfs.

chosts - Syntax: %chosts [-v]
This command allows you to toggle the select and de-selected status of processors. Only the selected processors will be used for immediate parallel processing.

The -v option gives the current Synergy connection status, it requires some extra time.

2.3) How to make remote processors listen

To activate remote processors, you will need to start one CID (command interpreter daemon) for each host. CID has a "server mode" and a "debugging mode". The debugging mode is capable of displaying outputs for all processes spawned by CID. Therefore, if you desire viewing the remote outputs, you should create individual windows on different hosts and in each enter:

```
%cid&
```

The command "sds" will start all daemons automatically in "server mode". Therefore there will be no output displayed at all. Note that for hosts that do not have PMD started, "sds" works slowly since CID must make sure there is no hanging PMD around before starting a new one.

CID will try to register itself with PMD (Port Mapping Daemon). CID will automatically start a new PMD if none is available at the local host.

To check for the remote processor accessibility from the local host, enter:

```
%cds
```

This command displays host availability for all "selected" entries in your local host file.

You can leave CID running even if the host is de-selected.

Both CID and PMD are user-privileged processes. They and all future processes generated by them must obey user restrictions associated with your login. Therefore, Synergy imposes no additional security threat to existing systems.

3 Writing Synergy programs

Building timing models before parallel programming can determine the worthiness of the undertaking in the target multiprocessor environment and prevent costly design mistakes. The analysis can also provide guidelines for parallelism grain size selection and experiment design (`http://joda.cis.temple.edu/~shi/super96/timing/timing.html`)

Except for server programs, all parallel processing applications can be represented by a coarse grain dataflow graph (CGDG). In CGDG, each node is either a repetition node or a non-repetition node. A repetition node contains either an iterative or recursive process. The edges represent data dependencies. It should be fairly obvious that CGDG must be acyclic.

CGDG fully exhibits potential effective (coarse grain) parallelism for a given application. For example, the SIMD parallelism is only possible for a repetition node. The MIMD parallelism is possible for any 1-K branch in CGDG. Pipelines exist along all sequentially dependent paths provided that there are repetitive input data feeds. The actual processor assignment determines the deliverable parallelism.

Any repetition node can be processed in a coarse grain SIMD (or scatter-and-gather) fashion. The implementation of a repetition node is to have a master and a worker program connected via two tuple space objects. The master is responsible for distributing the work tuples and collecting results. The worker is responsible for computing the results from a given input and delivering the results.

For all other components in the graph, one can use tuple space or pipe. The use of file and database (yet to be implemented) objects is defined by the application.

Following the above description results in a static IPC graph using passive objects. The programmer's job is to compose parallel programs communicating with these objects.

3.1) How to compose a Synergy program

In Synergy V3.0, a program only needs to interact with objects of three types:

a) tuple space
b) pipe

c) file
d) database (yet to be implemented)

An object must be opened first:

```
object_id = cnf_open("object_name", mode);
```

The mode is 0 for all non-file objects. For file objects, the mode can be one of: ("r", "w", "a", "r+", "w+"). Similar to file descriptors, the object_id will be used for object operations. Note that “object_name” is case-sensitive that MUST match the CSL description in the synthesis step.

For tuple space objects, you have three commands:

```
length = cnf_tsread(object_id, tpname_var, buffer, switch);
```

> It reads a tuple with a matching name as "tpname_var" into "buffer". The length of a read tuple is returned to "length" and the name of a read tuple is stored in "tpname_var". When "switch" = 0 it performs blocking read. A -1 switch value instructs a non-blocking read.

```
length = cnf_tsget(object_id, tpname_var, buffer, switch);
```

> It reads a tuple with a matching name as "tpname_var" into "buffer" and deletes the tuple from the object. The length of the read tuple is returned to "length" and the actual name of the read tuple is stored in "tpname_var". switch = 0 signifies a blocking get. A -1 switch value instructs non-blocking get.

```
status = cnf_tsput(object_id, "tuple_name",buffer,buffer_size);
```

> It inserts a tuple from "buffer" into the object of "buffer_size" bytes. The tuple name is defined in "tuple_name". The execution status is returned in "status" (0 means name collision or tuple value over-write).

For pipe objects, there are two commands:

```
status = cnf_read(object_id, buffer, size);
```

> It reads "size" bytes into "buffer". Status < 0 reports a failure. It blocks if the pipe is empty.

```
status = cnf_write(object_id, buffer, size);
```

It writes "size" bytes from "buffer" to the pipe.
Status < 0 represents a failure.

The use of pipes is similar to PVM and MPI (message passing interface) in semantics.

For file objects, we have the following commands:

- cnf_fgets(object_id, buffer)
- cnf_fputs(object_id, buffer)
- cnf_fgetc(object_id, buffer)
- cnf_fputc(object_id, buffer)
- cnf_fseek(object_id, pos, offset)
- cnf_fflush(object_id)
- cnf_fread(object_id, buffer, size, nitems)
- cnf_fwrite(object_id, buffer, size, nitems)
- cnf_close(object_id)

These commands assume the same semantics as ordinary Unix file commands except that the physical file name and location are transparent to the programmer.

The use of pipe and file objects requires no special programming. For compute-intense applications using tuple space objects some programming discipline is necessary:

3.1.1) Parallel worker termination

A parallel worker program assumes a simple "get-compute-put" cycle. For load balance and fault tolerance, the "get" is typically fetching a tuple with a certain pattern in its name. Since workers will loop forever until being told to go away, the problem is how to inform workers the termination for a graceful shutdown.

Synergy tuple space object assumes a FIFO order (this will be preserved in the full SLM implementation). This permits the use of sentinel for terminating work assignment tuples. As long as the worker returns the termination sentinel before it exists, all other workers will follow suite for a graceful shutdown. This will leaves one tuple in some objects after the termination. The Synergy runtime system will automatically removes these objects.

3.1.2) Load balancing

For a multiprocessor system, there are two types of load balancing requirements:

a) Parallel application initiation time; and
b) Runtime load balancing.

The first type requires finding the least loaded nodes for processing a parallel application. This determines the processors used for the application. The second type adjusts the dynamic load distribution thus has a large impact on how fast the application will be processed.

In Synergy, the first type load balancing is manually done. This means that the user must decide a set of nodes to use before executing the "prun" command. This step will be automated in the SLM implementation.

The second type load balancing requires adjusting processing grain size. There are at least three different techniques:

a) Optimal Fixed-Chunking
b) Guided self-scheduling.
c) Factoring

3.1.2.1 Optimal Fixed chunking [3]

Each tuple (work load) contains the same work size measured by number of data items to be processed (grain size). Assuming all data items carry the same workload, the optimal performance can be achieved if the grain size is

$$\frac{N}{\sum_{i=1}^{P} P_i},$$

where N is the total number of data items to be processed, Pi is the estimated processing power measured in normalized index values [3] and P is the number of parallel workers.

> For example, if N=1000, P_i=1 (all processors of equal power), P=10, the optimal grain size is 100. On the other hand, if P_1=1, P_2=2 and P_3=4, and P=3, the optimal grain size is 1000/(1+2+4)~=143.

Since the measures of Pi's are never accurate, it is difficult to approach the optimal using this algorithm without practical experiments.

3.1.2.2 Guided Self-Scheduling (GSS) [1]

Assuming there are N data items to be computed and P parallel workers, GSS tuple sizes are calculated according to the following algorithm:

```
R0 = N. /* Ri: ith remaining size */
Gi = Ri/P = ((1-1/P)i * N/P). /* Gi: ith tuple size */
Ri+1 = Ri - Gi.
Until Ri = 1.
```

For example, if N=1000, P=2, we have tuples of the following sizes:

```
500,250,125,63,32,16,8,4,2,1
```

Experimentations indicate that GSS puts too much work in the beginning. It performs poorly when employing processors with large capacity differences or computing problems with large computing density variances between repetitions.

3.1.2.3 Factoring [2]

Assuming there are P parallel workers, a threshold t>0 and a real value (0<f<=1), the factored grain sizes are calculated as follows:

```
R0 = N.
Gi = Ri * f / P
Ri+1 = Ri - (P*Gi)
until Ri < t.
```

For example, if N=1000, P=2, f=0.5, t=1, we have the following tuple sizes:

```
250,250,125,125,63,63,32,32,16,16,8,8,4,4,2,2,1,1
```

Since different program input can significantly change the work distribution using the same tuple size calculation algorithm, the presence of a "knob" (0<f<=1), gives the programs much latitude to adapt to both input and resource status changes.

3.1.2.4 Automatic Runtime Load Balancing

You can automate load balancing by adapting the tuning parameters in your application controlled by the parameters in the .csl file description (factor and threshold clauses). If automatic load balancing using fixed chunking is desired, a work load calibration process must be coded in both the master and the worker. Processor and network calibration will cost some extra time. The savings, however, are often worth more than the losses. (see $SNG_PATH/apps/ssc/albm for a coding example).

The packing and unpacking of working tuples are often the troubled spots in practice. Calculate your indices carefully. You can send or receive structures, as long as they are interpreted exactly the same by both parties.

3.1.3) Fault tolerance

In Synergy 3.0, worker fault tolerance does not require special programming. However, if the master sends the termination tuple BEFORE completing result collection, the application may not complete correctly. Premature worker termination is the cause of problem.

3.1.4) Naming

Note that the object types are declared in an application configuration file (CSL). The CSL file builds the parallel application by connecting the user defined object symbols and by defining program/object-to-processor bindings. Therefore, the object names defined in the CSL file must match the names used in the “cnf_open” statements EXACTLY. The binary names must also match actual file names exactly.

3.2) How to compose a CSL file

CSL stands for "Configuration Specification Language". Each application needs a configuration file to run. The CSL file controls the resource management, binary code management and program-to-program communication management.

Processor assignment is the resource management. Program path definitions are binary code management. Selecting and connecting programs with objects are process coordination management. Process coordination is the only necessary specification for an application. If all binary programs are placed in the $HOME/bin directory and no special resources are required, we can automate other management functions.

An example MIMD parallel application CSL file is follows:

```
================================================
Configuration: msort;

F: infile = sort.dat
        -> M: split
        -> F: F1 (type= pipe), F2 (type= pipe);

F: F1
        -> M: Sort1 = sort (exec_loc=argo.cis.temple.edu::/u/shi/apps)
        -> F: OF1 (type=pipe);

F: F2
        -> M: Sort2 = sort (exec_loc=zoro.cis.temple.edu::/u/shi/apps)
        -> F: OF2 (type=pipe);

F: OF1, OF2
        -> M: Merge

        -> F: out = sorted.dat;

/* The Synonyms */
S: F1, sort1.in,split.out1;
S: F2, sort2.in,split.out2;
```

```
S: OF1, sort1.out, merge.in1;
S: OF2, sort2.out, merge.in2;
=========================================================
```

In CSL, M stands for "module" and F stands for "passive object."

This CSL file describes a parallel mergesort network with the following processor assignments: (split, default), (sort1, argo), (sort2, zoro), (merge, default) as defined in the "exec_loc" clauses. A default processor is the computer where the CSL file is processed.

Note that the left-hand-side (LHS) of "=" defines the object name and RHS defines the physical entity name. For example, program modules "sort1" and "sort2" activates the same binary named "sort" The file object "infile" refers to a physical file named "sort.dat".

The path specification in the exec_loc clauses indicates the locations of the binaries. Its absence implies the use of $HOME/bin. We strongly encourage the use of $HOME/bin since it is the easiest to manage.

The S statements (synonyms) connect variously user named object symbols (in individual programs) to real objects. For example, object F1 is referred in program "sort1" as "in" but in program "split" as "out1".

If you draw on a paper the connections between M nodes, you will see a static SIMD process graph with two parallel sorters.

Here is a dynamic coarse grain SIMD example:

```
============================================================
Configuration: Fractal;

M: master = frclnt
        (factor = 50          /* Scheduling parameter definitions */
         threshold = 1
         debug = 3)           /* Module level debug switch, overwrites */
        -> F: coords (type= ts)
        -> M: worker=frwrk (type = slave)
        -> F: colors (type=ts)
        -> M: master;
============================================================
```

In this example, the processor assignments are to be automatically done by the system. It will use as many processors as selected by the "chosts" command. This system also assumes that the object names in the respective cnf_open statements of the parallel programs agree with this specification. Therefore we do not need synonym statements.

The "factor" and "threshold" clauses are for load balancing using the factoring method. The program "master" (binary name: "frclnt" uses "cnf_getf()" and "cnf_gett()" to obtain these values. The debug switch is for the object operators to show their operations by sending some buffer contents to the display. For details see $SNG_PATH/docs/sng_man.ps.

If you draw this configuration on a paper with actual processors instantiated, a "bag-of-tasks" graph or "scatter-and-gather" graph will appear.

If you absolutely hate typing, then wait for the full SLM implementation. It will have a much simpler configuration method.

3.3) How to compile Synergy programs

The Synergy passive operators are provided in a static link library named libsng.a. Linking to the library is the only requirement for making a Synergy parallel program:

```
% gcc program.c -o program -L$(SNG_PATH)/obj -lsng
```

If you want to debug your source, please supply the -g option.

Since your binaries are in the $HOME/bin directory, you can save a lot of management headaches by automatically moving the executables to that directory.

Note that compiling on a host only makes the binaries available to all hosts using the same file system. A separate compilation (or ftp, if binary compatible) is necessary for processors on a different file system (cluster).

4 Running Synergy programs

4.1) How to select/de-select hosts

You must decide two things before running a parallel application:

a) How many hosts ?
b) Which hosts ? (only necessary if the application requires special resource)

The first question can be answered by the timing model analysis. In practice, as we gain experiences, an "educated guess" often works as well.

The second question can be answered by evaluating the resource requirements of your application and their availability on the respective hosts, such as special licensed software packages, frame buffer, special device drivers, excessively large memory, disk, CPU cycles and network bandwidth needs.

You may wish to use manual processor assignments to satisfy the demands as much as possible. The manual processor assignments are done in the CSL file (see Application configuration). Note that using the manual processor assignments puts your application to the mercy of the availability of those designated processors. Failure of those processors will definitely bring down your application.

Once the decisions are made, you may choose your active hosts by entering:

```
%chosts [-v]
```

The -v (verify) option gives the current processor connectivity status. The command runs faster without the -v option.

4.2) Run

Simply enter:

```
%prun CSL_filename (without .csl extension) [debug | -ft]
```

In theory, prun should ONLY be entered from one of the “selected” hosts. However, a relaxed requirement is to have at least PMD/CID running on the local host. Otherwise, prun will automatically start them and quit. It will work the next time you enter prun. It will NOT force the local host be added to your “selected” host pool.

This command blocks the terminal until the application completes. You can run it in the background or put it in a shell script.

The -ft option activates the worker fault tolerance mechanism.

The [debug] option creates a parallel debugging session by generating a .ini and .dbx files for each program. You must keep the .ini and .dbx files with corresponding (-g compiled) binary and source to use the debuggers successfully.

4.2.1) Debug

Debugger is started automatically for each program by entering:

```
%<program_name>.dbx
```

If your local debugger is not called "dbx," edit the respective .dbx files to reflect the difference. You may send me an email, I can make you a special version.

To terminate a debugging session, enter:

```
%<application_name>.end
```

This cleans up all debugging scripts and the distributed objects.

4.2.2) Trouble Shooting

Unless you're extremely lucky, the very first time you use prun, the "verify" process will tell you that it is unable to execute process so and so. Several places you should check:

a) If the executable paths in csl file really exist. (Note again that unless you use NFS/AFS, you will have to make the executables once on every host.)

b) If you did not specify the paths, make sure that your binaries are copied to the $HOME/bin directory. The most often mistake is the absence of $HOME/bin directory. The Unix cp command actually copies all binaries to a file named "$HOME/bin"!

c) If you used specific processor assignments, make sure if $HOME/.sng_hosts includes all the machine names your application needs, especially the machine with the "cannot execute" process.

d) If all above failed to improve the situation, your cid's may be corrupted. Find the hosts where the programs cannot be started and restart a fresh cid by typing:

```
% cid &
```

f) If (e) could not improve the situation, resources are most likely low on selected hosts. Select a different processor and try again.

This should resolve most problems.

When the system resources become tight, the remote cids will refuse to start your programs and prompt you the "suspected resource problem ..." message. Choosing a different processor set is the most cost effective solution at all times.

The use of prun on any host makes this host the console of the launched application, since there will be a DAC running at this host. If you do not have CID/PMD running, prun will automatically start one for you.

4.3) How to monitor and control multiple parallel applications

Use the following command to monitor all running parallel applications associated with your login:

```
%pcheck
```

Pcheck is a generalized Unix "ps" command in that it has two monitors: an application monitor and a process monitor. It keeps track of all applications started from the local host. It also keeps track of all processes for all running applications initiated by your login.

You can inquiry and kill an application or a process. In case of killing an application using many processors, please be patient. The monitor may seem hanging for a while. Note that killing "pcheck" will not terminate any applications nor will it damage your running programs.

Pcheck uses only a single ASCII window. Therefore, you can monitor your parallel programs through a dialup line if necessary.

4.4) Clean up after troubled runs

A running parallel application utilizes many processes on many hosts. In a distributed environment, sudden processor re-boots and load changes are inevitable. To the parallel users, some processes can die suddenly or be swapped out causing extended timeouts and eventually application crash.

Killing a parallel application involves removing all related program and objects for an application. The best way is to use "pcheck". However, there will be cases when "pcheck" can hang for a long time.

In these cases, you should first kill "pcheck" and then try to kill the application specific DACs. You can use the Unix "ps" command to find DAC process id. Killing DAC should clean up all related programs and objects for this application if all threads are still controllable by this DAC.

The worst case is that you have to kill individual processes one host at a time. The script KILL was created to make it easier:

```
%KILL string
```

This command kills all processes with a display in "ps" that matches "string". This is similar to Linux "killall" command.

5 Example applications

The $SNG_PATH/apps directory contains three sub-directories: a) selftests, b) ssc; and c) nssc. The selftests sub-directory contains test applications for the three supported passive objects. Successful runs of these tests ensures the basic functions of the Synergy system.

The ssc sub-directory contains all solution space compact (ssc) applications. For an ssc application, linear or superlinear speedup is only possible when most of the sequential timing includes extended resource seeking times, such as swapping. In other words, these applications will have at the best sublinear speedups if uni-processor resource constraints are not a factor (such as cpu, memory and disk quotas).

The nssc sub-directory contains all non-ssc applications. Superlinear speedups are possible for these applications even if uni-processor resource constraints are absent. The secrete to obtain a high order superlinear speedup is to find an input that would force the worst case performance of the sequential program. The best recorded speedup was 1800 using 6 processors for the sum-of-subset problem. NSSC problems are mostly NP-complete problems.

6 Where to get Synergy

Follow this link: `http://spartan.cis.temple.edu/synergy` .

7 How to install

7.1) Unpacking

To uncompress, at Unix prompt, type

```
    % uncompress synergy-3.0.tar.Z
```

To untar,

```
    % tar -xvf synergy-3.0.tar
```

A directory called "synergy" will be created and all files unpacked under this directory.

7.2) Compilation (for source code version only)

To compile, type

```
    % make
```

The current version has been tested on these platforms:

- SunOs 5.9 (Blade 500)
- RedHat Linux Release 9 (Strike).

The makefile will try to detect the operating system and build binaries, libraries and sample applications. You may need to edit the makefile if your system requires special flags, and/or if your include/library path is nonstandard. Check the makefile for detail.

The -ft option is now stable and tested on both SunOS 5.9 and RedHat 9. It is recommended for production runs. The user is warned that the automatic fail over mechanism can confuse your programmed logic if used in debugging sessions.

For technical correspondence, please write to shi@temple.edu.

8 Summary

This document describes the general philosophy of the Synergy design and principles of the coarse-to-fine grain parallel programming method. We assumed that efficiency is of the primary concern here. For some applications, however, speedup may be a higher priority. For these applications, MPI/MPICH may be a better platform. In all cases, analysis using timing models can provide an effective guide to your performance dreams.

Synergy V3.0 represents a unique union of theory and practice for parallel processing using multiple networked high performance processors. It includes the essential tools for making production quality parallel programs.

A new version of Synergy is under development. The new system will provide utilities for more effective computational as well as transactional processing. Efficiency and fault tolerance are among the highest priority in the design objectives. For latest news, please check our WWW home page at http://spartan.temple.edu/synergy.

==

9 Acknowledgments

I would like to thank the Technology Transfer Office of Temple University for the financial and spiritual support since 1990. Thanks also go to the International Business Machines Corporation for the fruitful experiments conducted in 1990-1992 period. Thanks for the former Digital Equipment Corporation for the funding of the Hermes Financial Simulation Laboratory at the Wharton School of University of Pennsylvania and for the loan of a DEC Alpha processor at the Supercomputing'94 Conference. Support from National Science Foundation for supercomputer accounts and time at Cornell Theory Center, Pittsbugh Supercomputing Center and San Diego Supercomputing Center have also helped developing the theories described in this document.

I would also like to thank my students for their enthusiasm and diligence in the course works of CIS750, CIS669, CIS673/615 and various independent study, Ph.D and Master's projects. In fact, all example applications are from these courses.

Thanks also go to the Temple University and CIS networking and computing support staff members for maintaining an excellent distributed computing environment.

10 References

1. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers," IEEE Transactions on Computers, C-36, 12 (December 1987), 1425-1439.
2. S.F.Hummel, E. Schonberg and L. E. Flynn ., "Factoring -- A Method for Scheduling Parallel Loops," CACM, Vol., 35, No.8 (August 1992), 90-101.

3. Yuan Shi, “A Distributed Programming Model and Its Applications to Computation Intense Problems for Heterogeneous Environments,” AIP Conference Proceedings 283, Earth and Space Science Information Systems, 827-848, 1994.
4. Piranha: D. Kaminsky, "Adaptive Parallelism with Piranha," Ph.D Dissertation, CIS Department, Yale University, 1994.
5. PVM: V.S.Sunderam, "PVM: A framework for parallel distributed computing," Practice and Experience, 2(4):315-339, December 1990.
6. Message Passing Interface Forum (http://www-unix.mcs.anl.gov/mpi).
7. Yuan Shi, “Program Scalability Analysis,” (http://joda.cis.temple.edu/~shi/super96/timing/timing.html)

===

End of Document

===