A QUICK INTRODUCTION TO SYNERGY V3.0+

Justin Y. Shi (aka Yuan Shi)
shi@temple.edu
(215) 204-6437 (Voice)
(215) 204-5082 (Fax)

(c)Temple University
Philadelphia, PA 19122
January 1995
(Revised March 2004)
(Revised February 2013)

==

Index:

==

1) Preface

As the driving force for single processor speed improvements comes to a stop, it has become clear that future exascale computers must rely on increasingly larger numbers of processing and networking elements. Unlike traditional HPC architectures that focus on maximal deliverable performance, the research focus must shift to harness volatile computing and communication resources of extreme scales.

Parallel programming is also a long standing research challenge. There is no lack of controversies on the subject matter. The current main-stream programming paradigms are explicit parallel, such as MPI (message passing interface) and OpenMP (shared memory).

A 2012 accidental discovery has made the long hidden First Principle come to surface: Statistic Multiplexed Computing (SMC). Statistic multiplexing is a proven First Principle for harnessing massive not-so-reliable components for extreme scale communication systems. The performance and reliability of the Internet today are not a result of incremental improvements based off the established circuit-switching technology but a paradigm shift to packet switching technology. The ability to harness not-so-reliable components has also made it easier to integrate circuit-switching components. Since there is no physical scalability limitation of the First Principle, except for IP addressing challenges, the Internet architecture today is poised to scale indefinitely into the future.

The Synergy project was an early attempt of applying the First Principle to high performance computing. The project started in 1982 based on my Ph.D. dissertation at the University of Pennsylvania of Philadelphia, USA. The original challenge was to hide the national security sensitive insights from a global econometric simulation project that requires all countries to participate. The project was named LINK, led by the Nobel Laureate Dr. Lawrence Kline at the Economics Department of Wharton School of University of Pennsylvania. Data parallel computing was a natural proposed solution. My dissertation (supervised by Dr. Noah Prywes) was to find a solution to the distributed termination challenge: with only the knowledge of publically exposed data, how to ensure the accuracy of global simulation results since the world econometric simulation cannot progress until all countries are convergent on their own economies.

The solution was very simple. It boils down to approximately less than a dozen lines of protocol code for solving the synchronized distributed termination problem. Retrospectively, it was the first sight of the First Principle. For many years, we knew the importance of harnessing massively many components but there was no "smoking gun" to criticize explicit parallel methods, such as MPI and OpenMP, until 2012.

It turned out that all practical TCP/IP implementations are the sources of growing application instability due to the use of explicit parallel programming paradigms. This is because the lower (media) protocol layers (1-4) contain error checking and re-transmissions to mitigate transient component failures. The higher (host) layers do not have such protection; the explicit parallel programming paradigms also assume reliable application-level communication. Thus, every transient component error becomes a single-point failure for a parallel application, similar to the circuit switching networks. Bigger applications having more failure points are more likely to fail. The current MTBF (mean time between failures) of a parallel application using more than 1024 processors has fallen to less than 60 minutes.

The Synergy project attempts to apply the First Principle in scalable networking to parallel computing. It uses the Tuple Space programming paradigm, an implicit data parallel paradigm, to allow SMC architectures.

Since implicit parallel programs are inherently simpler than explicit parallel programs, we (with Feijian Sun) have also studied the possibility of automating the generation of data parallel programs based on a Parallel Markup Language (PML). This release (Synergy3.0+) includes the basic Synergy core and the prototype of PML extension.

The material disclosed in this document is part of two patents (U.S.#5,381,534 and #5,517,656) and a provisional patent filed in 2008: "Fault Tolerant Self-Optimizing Multiprocessor System and Method Thereof". Parallel processing researchers, faculty members and graduate students are welcome to a free evaluation copy of Synergy3.0+. 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 prototype of Statistic Multiplexed Computing (SMC) system for parallel applications. It is the converged solution from earlier Passive Object Flow Programming (POFP) and Stateless Machine (SLM) research. The Synergy 3.0+ implementation, however, carries the traces of POFP and SLM projects. The implementation is not as scalable as the would-be full implementation of SMC architecture. But the Tuple Space programming paradigm allows teaching and demonstrating the benefits of applying the First Principle even using the rudimentary implementation.

In Synergy3.0+, a parallel application needs to use passive objects for inter-process(or) communications. There are three passive object types: Tuple Space, File, and Pipe. The Pipe object is deprecated. A parallel application consists of data parallel programs connected via these named passive objects. For a compute intensive application, the typical organization is the master-tuple space-worker configuration. Each master and associated workers corresponds to a single compute intensive core in the sequential program, iterative or recursive. The user is responsible for creating the data parallel master and worker programs via tuple communications. Each application must also have a configuration file specifying the relationships between components. The Synergy runtime requires the configuration (.csl) file to tie the parallel programs and passive objects into a coherent parallel application at runtime.

Each Synergy program, master or worker, is a localized stateless program that it can run on any of the distributed computers in a given cluster. Unlike the explicit "maximalist" parallel programs that a single program body contains both the master and work's logic, the Synergy master only concerns with data distribution and result collection. The worker only concerns with fetching unfinished data; compute the intensive core and result submission. This makes Tuple Space-based parallel programming similar to sequential programming, except for the data parallel flows. This was the basis for our PML research.

The File object was designed for applications that wish to maintain a constant source but moving file locations. Since the First Principle is for harnessing massively many not-so-reliable resources, the Synergy paradigm also allows integration with legacy MPI and OpenMP codes.

The current Synergy 3.0+ tuple space objects will converge to a single global distributed tuple store (Synergy 4.0) implemented via a Unidirectional Virtual Ring (UVR). Therefore, there is no plan to offer dynamic tuple space objects.

1.2) Historical Background

Data parallel computing is a well-studied subject in 1970's. Early dataflow machines had demonstrated the feasibility of automatic formation of all parallel processing patterns in fine grain parallel experiments. Harnessing volatile resources was not in the agenda. Even though it was thought that higher degree of parallelism could deliver better performances, it was evident early on that this was not the only factor in dataflow application experiments. Due to the lack of quantitative parallel performance evaluation tools and multiple other factors, early dataflow machines failed to deliver satisfactory performances. The single processor speeds had quickly surpassed the fine grain dataflow machines.

The lack of quantitative parallel program evaluation tool played a positive role in the pursue of extreme scale HPC application. The explicit paradigms have produced breath-taking results as recorded by the Top500.org site. The rapidly shrinking application MTBF is not well reported. But the research communities are well aware of the problems.

The Tuple Space paradigm forces the separation of functional programming from process coordination and resource management functions. In comparison, explicit parallel programming systems have made these functions responsibilities of the programmers. This has made parallel programming unnecessarily complex.

The passiveness of inter-program communication and synchronization (IPC) objects confines dynamic application dataflow to a static, bipartite IPC graph. In Synergy, this graph is used by the runtime system to deliver process coordination and resource management functions.

Synergy V3.0+ uses this static IPC graph and the dynamic application dataflow to automatically map parallel programs the cluster of networked computers. The automatic processor assignments are afforded by the First Principles of SMC and data parallel processing.

Using the static IPC graph also revealed that the application's communication volume is a function of processing granularity which can be optimized only at runtime by knowing the application-dependent processing capabilities. This revelation helped the development of Timing Model method (http://spartan.cis.temple.edu/shi/public_html/super96/timing/timing.html) for quantitative performance analysis of parallel applications.

Once the processor assignments are determined, Synergy V3.0+ automatically generates the statistic multiplexed computing layer (DAC for Distributed Application Controller) that translates the user domain symbols into execution environment parameters, such as IP (Internet

Protocol) addresses and port numbers. This layer allows the application to harness the volatile computing and communication resources at runtime. In Synergy V3.0+, DAC is automatically removed after the application terminates.

Fault tolerance is a natural benefit of the SMC architecture. A compiled Synergy application can use any number of parallel processors at any time. Processor faults discovered immediately before a run are automatically isolated. Processor faults during a parallel execution are protected by "shadow tuples". Synergy V3.0+ can automatically recover the lost tuples from the faulty workers with very little overhead. There is no special programming required to benefit from this feature. Master fault tolerance still requires checkpoint-restart (CPR). The Synergy application has reduced the size of CPR from O(P) to O(1). This can result significant performance gains and energy savings.

Another Tuple Space paradigm benefit is the ease of performance tuning. Unlike explicit parallel programs, Synergy programs are fully decoupled in time and space (even more so than the Linda programs since Synergy does not bind tuples to processes at compilation time). Performance tuning (full overlapping of computing and communication times) is possible by fine tuning the parallel processing granularity in the target processing environments.

These granularity adjustments can also be automated using a number of methods, such as fixed chunking, factoring and others (see $SNG_PATH/apps/albm as an example).

Synergy V3.0+ runs on clusters of distributed memory Unix workstations.

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

1.2.1) Synergy vs. MPI and OpenMP

MPI and OpenMP are two explicit parallel programming paradigms based on distributed memory and shared memory models. As discussed earlier, the tight binding between program and data has made it impossible to inject the SMC layer into the parallel application by the First Principle. Conversely, it is possible to integrate multiple MPI and OpenMP programs for exascale applications by the First Principle.

1.2.2) Synergy vs. Linda

Linda uses a virtual global tuple space implemented using a compile time analysis method. It supports typed tuple space operations. The main advantage of the Linda method is the potential to reduce communication latency. This is because many tuple access patterns can be un-raveled into single lines of communication. Thus the compiler could build the machine dependent codes directly into the sender without going through an intermediate runtime daemon that would potentially double the communication latency of each tuple transaction.

This perceived advantage prohibits the introduction of the First Principle. Therefore, Linda programs cannot benefit from automatic data parallel processing and cannot tolerate volatile environments.

1.2.3) Synergy vs. Dataflow Machines

The early dataflow machines did not have the concept of the First Principle. The only focus was parallel performance. The lack of quantitative parallel program analysis tools also has prevented the stake holder to discover the granularity problem soon enough. In comparison, Synergy programs allow the runtime application of the First Principle. The granularity can be tuned dynamically without program recompilation. Thus we hope that it is possible for it to gain wide acceptance.

---

2) Installation

a) Download from http://spatan.cis.temple.edu/synergy
b) Untar and unzip to your local installation path, say /usr/local/synergy. There is also a package for sample applications.
c) Change to /usr/local/synergy, enter “make”. That’s it.

The current release was compiled under Ubuntu 10.4, Redhat 6, Solaris 10. Please email shi@temple.edu for support of other systems.

If you have logged in as a cluster user and the Synergy.tar.gz was installed on the shared mount point, then you may only need to configure your account once. The cluster will automatically replicate your setting to all nodes on the cluster. Otherwise, you will need to install one copy on each of the communicating file systems and configure the nodes so they can “see” each other.

3) User Account Configuration

In the following, we assume "synergy_dirctory" is the Synergy installation path. Typically it is /usr/local/synergy for all users in the cluster or your personal path: ~/synergy for a private HPC cloud. We assume that you have logged in as a cluster user (with identical UID on all nodes in the cluster).

We also assume the default shell is csh, tcsh or bash. Other shells can also be supported by minor changes to the following illustration.

3.1) Local environment setup

Edit your shell RC to add the following:

```
Csh/tcsh(.cshrc):
      setenv SNG_PATH synergy_directory
      set path=($SNG_PATH/bin . $path)
bash(.bashrc):
      SNG_PATH = synergy_directory
      PATH = $PATH:$SNG_PATH/bin:.
      export SNG_PATH
```

To activate these definitions, enter:

```
      % source .cshrc or .bashrc
```

To check, you can enter:

```
      %which cds
```

It should points to your Synergy installation directory/bin/cds. You are good to go.

3.2) How to Create a processor pool:

Setup a local processor pool file is only 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.

                  Ex1: %shosts 155.247.177.1 129.32.32.100
                  This will create a host list with all hosts of
                  address 155.247.177.* and 129.32.32.*.

                  Ex2: %shosts default
                  This creates a host list with the same address prefix
                  as the current host.

                  Note: 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.

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 parallel processing
                  when an application is run.
                  The -v option gives the current Synergy connection
                  status, it requires some extra time.
```

3.3) How to make remote processors participate

To acquire remote processors to participate in the same parallel application, you will need to start one “pmd” (port mapper daemon) and one "cid" (command interpreter daemon) for each of the nodes. You can do this manually by “ssh” to the remote nodes. This gives the option to view runtime printfs from the application(s).

The script “sds” was made available to start these daemons automatically according to your ~/.sng_hosts file. However, due to changes in “ssh” (secure shell), the command “sds” may need tweaking before working reliably. The location of the “sds” script can be found in ~/synergy/bin.

```
      %pmd&
      %cid&
```

Once all daemons are started, to check for the remote processor accessibility from the local host, enter:

```
      %cds
```

Both cid and pmd are user privileged processes. They and all future processes generated by them can only do what your account(s) can do according to your security permits and restrictions. Therefore, Synergy imposes no additional security threat to your existing systems.

==

4) Writing Synergy programs

Building timing models before parallel programming can prevent costly design mistakes. The analysis results provide guidelines for parallelism grain selection and experiment design (see $SNG_PATH/docs/smodels.ps).

After analyzing many programs assuming the use of current and future communication and computing devices, it has become clear that any parallel program, except for the loop-for-ever type programs, 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 a loop or a recursive process. The edges represent data dependencies. It should be fairly obvious that application dependent IPC graph must be acyclic.

The IPC graph 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. Pipelines exist along all sequentially dependent paths provided that there are repetitive input data feeds. The actual processor assignments and the application dataflow determine the deliverable parallelism types.

Any repetition node with an independent kernel can be processed in a coarse grain SIMD (or bag-of-tasks) fashion. This further decomposes a repetition node into 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.

The above description results in a static IPC graph using passive objects. The programmer's job is to compose parallel programs interfaced with these objects. For more detailed discussion, see $SNG_PATH/docs/sng_man.ps.

4.1) How to compose a Synergy program

```
---------------------------------
```

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

```
                      a) tuple space
                      b) file
```

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+"). The object_id is for future accesses.

For tuple space objects, we 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
            indicates 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 return in "status"
      (0 means name collision or over-write).
```

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)
```

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

The use of tuple space objects needs some discipline:

a) Parallel worker termination

A typical parallel worker program has a simple "get-compute-put" structure. Since the Tuples have no particular order, the challenge is to let all workers terminate gracefully when all work is done.

The Synergy tuple space object assumes a FIFO order. This means that Tuples issued by the same master will inherit the issuing order when the Tuples are retrieved. This permits the use of sentinel for terminating parallel workers. Since a worker's main working cycle is "get-compute-put", the termination sentinel can be returned before a worker exists. This protocol always leaves one tuple in the Tuple Space object after the termination of the application. The Synergy runtime system automatically removes these objects. (see $SNG_PATH/apps/ssc/fractal for example)

b) Performance Optimization

Unlike explicit parallel programs, the Synergy master program must distribute the working tuples by certain granularity. This can be done in fixed size or according to some automatic tuning algorithm, such as factoring, fixed chunking or guided self-scheduling (GSS).

Experiments show that optimized Synergy application can consistently out-perform explicit parallel programs in volatile environments even without failure. The explicit parallel programs are best suited for dedicated environments without resource volatility.

If a manual method is chosen, such as factoring or chunking, the scheduling parameter(s) must be adjustable without changing the code. (see $SNG_PATH/apps/ssc/matrix for example)

If automatic performance optimization using fixed chunking is desired, a work load calibration process can be coded in both the master and the worker. Processor and network calibration will cost some extra time. For applications with long running cycles, the savings can be substantial. (see $SNG_PATH/apps/ssc/albm for a coding example).

The packing and unpacking of Tuples are probably the most difficult challenge for Synergy programming. Frequent mistakes are wrongly computed indices. But these can be easily debugged.

c) Fault tolerance

Synergy V3.0+ comes with worker fault tolerance support. As a standard practice, the master should only send the termination sentinel AFTER completing the result collection. Otherwise, the death or a slow worker can hang the entire application. (see $SNG_PATH/apps/ssc/matrix)

The best way to learn how to write a Synergy program is to check out all examples in the $SNG_PATH/apps directory.

Note that the object types are declared in an application configuration file (CSL). The CSL file defines the parallel application by connecting the user defined objects and programs. The CSL file instructs the Synergy runtime to generate the customized statistic multiplexing middle layer automatically.

4.2) 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: frwrk1=frwrk (type = slave)
      -> F: colors (type=ts)
      -> M: frclnt;
```

==

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 performance optimization using the factoring and fixed chunking methods. 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.

4.3) How to compile Synergy programs

The Synergy passive operators are provided in a linkable object library. 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.

Note that in a cluster environment, compiling on any host makes the binaries available to all hosts sharing the same file system. A separate compilation is necessary for nodes residing different file systems. They too, can participate in the same application.

==

5) Running Synergy programs

5.1) How to select/de-select hosts

The processor pool that you have setup for your account defines the maximal range of resources available to your applications. You can decide two things before running a parallel application:

a) How many hosts for this run?
b) Which hosts? (This is only necessary if the application requires special resource)

The first question can be helped by the timing model analysis (see $SNG_PATH/docs/tmodels.ps). 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, very large memory, very large disk, GPUs and special networks.

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. If redundancy is not available, 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.

5.2) Run, debug and monitoring

Simply enter:

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

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.

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. The debugging process follows the similar routine as debugging a sequential program, provided that you typically need to start two debugging sessions at the same time: one for the master and the other for the worker. You can examine the Tuple Space operations in detail, step by step.

To terminate a debugging session, enter:

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

This cleans up all debugging scripts and the distributed objects.

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. If your nodes reside on multiple file systems, you will have to copy the executable manually to every file system.

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 have failed to improve the situation, your cid's may be in a stale state. Find the hosts where the programs cannot be started and restart a fresh cid by typing:

```
% cid &
```

e) If (d) still cannot improve the situation, resources may be low on selected hosts. Select a different processor and try again.

The above steps should resolve most common 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.

To monitor the parallel execution, enter:

```
%pcheck
```

Note: The use of “prun” on any host makes this host the console of the launched application. All subsequent operations, such as “kill” in particular, for this application should ALL be performed from this host.

5.3) How to monitor and control multiple parallel applications

Use the following command to monitor your running parallel applications:

```
%pcheck
```

Pcheck is a generalized Unix "ps" command for parallel applications. 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.

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 hang for a while. Note that killing "pcheck" will not damage your applications.

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

5.4) Special cleanup procedure

A running parallel application utilizes many processes on many hosts. In a large scale cluster, sudden processor re-boots and load changes are inevitable. To the applications, some nodes 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 the application. The best way is through pcheck. However, there will be cases when process threads are lost. This will happen when the majority processors have low resource levels. Pcheck may hang for a long time.

In these cases, we should first kill pcheck and then try to kill the application specific "dac" (distributed application controller) on your local host. You may use the Unix "ps" command to find the process id first. Killing dac should clean up all related programs and objects for this application if all threads are still controllable by "dac."

You will have to kill individual processes one at a time if dac has lost their threads. This case is rare but does happen once in a while.

==

6) 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.

===

7) Parallel Markup Language (PML)

One lasting curious question in the parallel processing research community is the capability to generate parallel programs automatically. For many years, the efforts fell short to meet the expectations. Most people have given up the hope. We believe that the application of the First Principle may resurrect the hope since data parallel programs are inherently simpler to compose compared to explicit parallel programs.

The PML work by Feijian Sun had pushed the technology envelope to a new height.

PML comes with a compiler. It is located in the PML subdirectory under the directory “synergy”. It requires Java to run. There are three essential directories: classes, conf and lib. To run the parallelizer:

```
%java parallelizer
```

PML marked application examples are under “apps/PML-Examples”. Each example comes with sequential program, PML marked sequential program and generated Synergy programs. Please review the README file before proceeding.

Before compiling the PML marked applications, there are two additional packages to be installed under the “synergy” directory:

a) ntsh: This is the extended Tuple Space daemon. It must be compiled to replace the running tsh.
b) api_lib: These are the extended Tuple Space API calls. They will need to be compiled into a static runtime library.

===

8) PML Samples

- Init: Basic extended Tuple Space function test.
- Laplace: A PML Laplace markup example
- Lapsig: A PML Labsig markup example
- LUFactor: A PML LU factorization markup example
- Nqueen: A recursive nqueens solver example
- Possibilities: Future possibilities
- Real: A PML matrix multiplication example
- Tsppar: A PML parallel TSP (traveling sales person) solver

===

9) Where to get Synergy

Follow http://spartan.cis.temple.edu/synergy

9.1) Unpacking

To uncompress, at Unix prompt, type

```
                  %tar xvfz SynergyV3.0+.tar.gz
                  %tar xvfz SynergyV3.0+Apps.tar.gz
```

A directory called "synergy" and a directory “apps” will be created.

9.2) Build Synergy Runtime (for source code version only)

To compile, type

```
                        % make
```

You need to complete the instructions in Section 3 to complete the setup to test the build.

The current version has been tested on these platforms:

- SunOs 5.9 (Blade 500)
- RedHat Linux Release 9 (Strike).
- Ubuntu 10.4

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 not yet stable. Please email me for if you encounter problems.

When multiple different operating systems or different versions of the same operating system are used at the same time, the tuple space daemons may go down. A total recovery may be necessary in these cases.

For technical correspondence, please write to shi@temple.edu. Due to limited resources, I cannot promise immediate responses. But I can always find time to fix bugs eventually.

==

10) Summary

This document describes the general philosophy of Synergy V3.0+ design and our efforts to apply the First Principle of harnessing volatile resources to parallel applications. Unlike traditional systems research, the practice of the First Principle to parallel computing requires applications API participation.

Like the packet switching paradigm shift, we also believe a similar paradigm shift is necessary for exascale parallel applications.

A new version of Synergy V4.0 is under design. The new system will provide full implementation of UVR, utilities for more effective computational as well as transactional data intensive applications. Harnessing volatile resources is the highest priority in the design objectives. For latest updates, please check our WWW home page at http://spartan.temple.edu/synergy.

==

11) Acknowledgments

The author thanks the National Science Foundation for the support of MRI (Major Research Instrument) grant #CNS0958854. This grant gave the Temple University’s computational scientists and students the opportunity to participate in cutting edge research and development activities within campus. This grant also gave us the opportunity to create the Temple Elastic HPC Cloud (TCloud) that has been used in graduate courses for the last two years. (https://sites.google.com/a/temple.edu/tcloud/)

The author also thanks the Technology Transfer Office of Temple University for the financial and spiritual support from 1990 to 2007. Thanks also go to the International Business Machines Corporation for the fruitful experiments conducted in 1990-1992 period. Thanks for the 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.

Thanks also go to my students for their enthusiasm and diligence in the course works of CIS750 (Graduate Seminar), CIS669 (Distributed and Parallel Processing), CIS673 (Analysis of Algorithms) and various independent study, Ph.D and Master's projects. All example applications are from these courses. The initial TCloud work was done with my student Moussa Taifi and CIS network administrator Tom Stauffer. The original Synergy code was created via multiple independent study courses under my supervision. Isaac Rajkumar had setup a solid code base for the many year efforts to continue. Feijian Sun’s work on PML and “shadow tuple” had pushed the technology envelope to a higher ground. At the time of this writing, Moussa’s website has the most up-to-date publications and technical resources for the TCloud (https://sites.google.com/site/moussataifishomepage/)

Thanks also go to Johnathan Wang, a former Temple undergraduate student then a graduate student at Yale University, for setting up the first gopher and WWW connections for the Synergy project.

==

End of Document

==

Contact: <shi@temple.edu>
http://spartan.cis.temple.edu/synergy