**Тема 10. Паралелно програмиране с ERLANG**

**(Приложения за симетричен мултипроцесоринг)**

**Мултипроцесорно програмиране:** За линейността (скалируемостта) и ефективността се изискава: 1) декомпозиция на приложението на подходящо множество процеси n>=p (по възможност n>>p); 2) избягване на страничните ефекти от конкурентния достъп до данните и от синхронизацията: взаимна блокировка (deadlocks), състезателен достъп (race condition) и т.н.; 3)в ERLANG страничните ефекти са почти елиминирани поради отсъствието на общи променливи и многонишково присвояване; 4) избягване на тесните места от последователната част на програмата; 5) при МП има къси съобщения и по-рядко обмен м/у процесите.

**ERLANG и мултипроцесорите:** Чертите на ERLANG, които подкрепят мултипроцесоринга са: конкурентност без критични области, асинхронен МП, прозрачно разпределение по ашини, аядра и виртуални възли, отказоустойчивост (чрез репликиране) и без mutex-и и транзактивна памет (освен ets/dets). BM се стартира в многоядрен режим със следните флагове: c:>erl –smp +S N, където –smp стартира ВМ в многоядрен режим – Symmetric Multiprocessing; +S N стартира ERLANG с N BM – всяка ВМ е нишка със самостоятелно планиране – но нишките имат общ процесен контекст. Нормално е N=>p, където p е брой ядра. N има стойност по подразбиране. Освен паралелни ВМ е желателно да се паралелизират и отделните приложения – по управлени, по данни и т.н.

**Паралелизъм с криптични области:** Общите данни м/у групата нишки на паралелното приложение намалява ефективността заради поддържането на консистентността м/у кешовете на ядрата. Подари това е трудно да се постигне ефективно работеща многонишкова паралелна програма на повече от 2-4 UMA-ядра. Всеки процес има собствен защитен контекст – за обмен се ползват променливите-ключалки – с произтичащите от това забавяния и странични ефекти. Процесите осигуряват истинска защита на паметта, на цената на значителен системен свръхтовар за планиращото ОС-ядро, което управлява достъпа до паметта и движението на процесорния контекст по слоевете на виртуалната памет. Затова обработката в ERLANG-машината се нарича „процес” – като самите ERLANG -„процеси” не са дори нишки от гледна точка на ОС, но ERLANG -машината ги управлява като комуникиращи последователни процеси, а ключалките са заменени от „съобщения” – за лесен пренос м/у мултикомпютри и мултипроцесори – или дори за съчетаване на двете архитектури.

**Паралелизъм без критични области:** Функциите на ERLANGнямат общи данни – освен предадените чрез аргументите на обръщението – т.е. подобно на RPC-MP. Именно това позволява миграцията на всеки „процес” на ново ядро или процесор или даже на нов възел – т.е. с отделно адресно пространство. Този модел на комуникации м/у процесите обаче е ефективен когато свързваните процеси нямат „много” голям общ контекст. Миграцията на голям общ контекст поражда неефективност от комуникационния свръхтовар.

**Паралелизиране с хеш:** Прилага се за паралелен (p2p) достъп до общ ресурс в хеш-таблици –distributed hash tables. Има разпределение на данните (като е в разпределителна система). Ключа определя в кой „възел” ще се съхраняват асоциираните с него данни. Обичайно възлите се рганизират в логически пръстен по свои n-разредни идентификатори и всеки нов компонент се асоциира с този възел, който е с най-близко ID до неговия ключ. Постига се квази-балансирано разпределяне по възли на контекста – по памет и по време за достъп за запис и писане.

**Общи данни: ETS/DETS:** [Disk] Erlang Term Storage са вградени системни модули за ускорен достъп до огромни по размер записи в/у диск или направо в ОП. Тези данни представляват подходящо структурирана за бърз достъп таблици с двойките ключ-стойност. Достъпът се влияе много слабо от размера на таблиците. Тези данни не са специализиран SV механизъм, но са много полезна алтернатива на МР при паралелните процеси, понеже позволяват ефективен конкурентен достъп при значителен размер общ контекст. Частично достъпът се ускорява поради това, че не е сериен както при „съобщенията”, макар че „съобщенията” м/у 2 паралелни процеса се разполагат в общата памет на възела, достъпът не ползва друг адрес освен началото на съобщението. При ETS данните съществуват в 1 копие, а при съобщенията се дублират – в областите на изпращащия и получаващия процес – което може да се разполагат в общата памет на възела. Данните позволяват множествен достъп, което е важно предимство при локално- и глобално-синхронните алгоритми, които изискват групови комуникации и то по възможност с паралелна имплементация.

**D/ETS операции: 1)**създаване на ETS: ets:new; 2)отваряне на DETS: dets: open\_file; 3) включване на двойка или списък двойки ключ-стойност: insert(Tablename, X); 4) извличане на стойност по ключ: lookup(Tablename, Key) – може да се върнат в списък повече от 1 стойност за някой типове таблици, при неоткрит ключ резултатът е празен списък; 5)прекратяване на ETS: ets: delete(TableId)-прекратяването е изтриване на данните с освобождаване на паметта, ERLANG не прави изчистване на неадресираните таблици, но ets се изтрива когато терминира създалия процес, а dets се затваря; 6)затваряне на DETS: dets:close(TableId).

**Типове D/ETS:** Има няколко типа таблици, те са: 1) set и ordered\_set: точно 1 обект на ключ без или със подреждане на обектите (напр.:{a,1} {b,2} {c,3}). Подреждането на обектите води до различна скорост на операциите. 2) bag и duplicate\_bag: повече от 1 обект на ключ – с едно или повече копия на всеки от обектите (напр.: {a,1} {a,2} {b,2} и {a,1} {a,1} {a,2} {b,3}). Съществува максимален брой таблици, поддържан от 1 ERLANG-възел.

**Паралелизиране на списък:** 1) последователно-рекурсивна (поелементна) обработка на списък с lists:map: map(\_,[])->[]; map(F,[H|T])->[F(H|map(F,T)]. 2)паралелна версия на map-pmap: pmap(F,L)->S = self();

Ref = erlang: make\_ref(), %make\_ref връща нов адрес;

Pids=map(fun(I)->spawn(fun()->do\_f(S,Ref,F,I) end)end,L),gather(Pids, Ref);

do\_f(Parent, Ref, F, I) -> Parent !{self(), Ref,(catch F(H))];

gather([Pid|T], Ref)->receive {Pid, Ref, Ret ->[Ret\gather(T,Ref)] end; gather([],\_)->[].

3) паралелната pmap: обработва като map, но за всеки елемент на списъка създава нов процес с do\_f и го маркира с Ref; 4) обработващите процеси могат да завършат в произволен ред, затова при събирането на резултатите gather избирателно ги сортира по Ref; 6) (catch F(H)) се използва вместо директно F(H) за да се прихванат изключенията в процесите-наследници, които да терминират родителя pmap.

**Гранулираност:** Паралелизирането на списъци може да подведе, че за ефективна паралелна програма е достатъчно да подменим map с pmap. За ефективност допълнително трябва да се подбере подходящата гранулираност. При pmap паралелизма P=l е максимален (l = length(L) е мощността на списъка), съответно гранулираността е най-фина – това е другият край на скалата спрямо последователната map. Трябва да се анализират междинните варианти в зависимост от времевата сложност TF на F(L). Ефективното приложение на pmap може да има само при TF >> ТSPAWN.